P&S Heterogeneous Systems

Parallel Patterns: Merge Sort

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

2 January 2023

Parallel Patterns

Reduction Operation

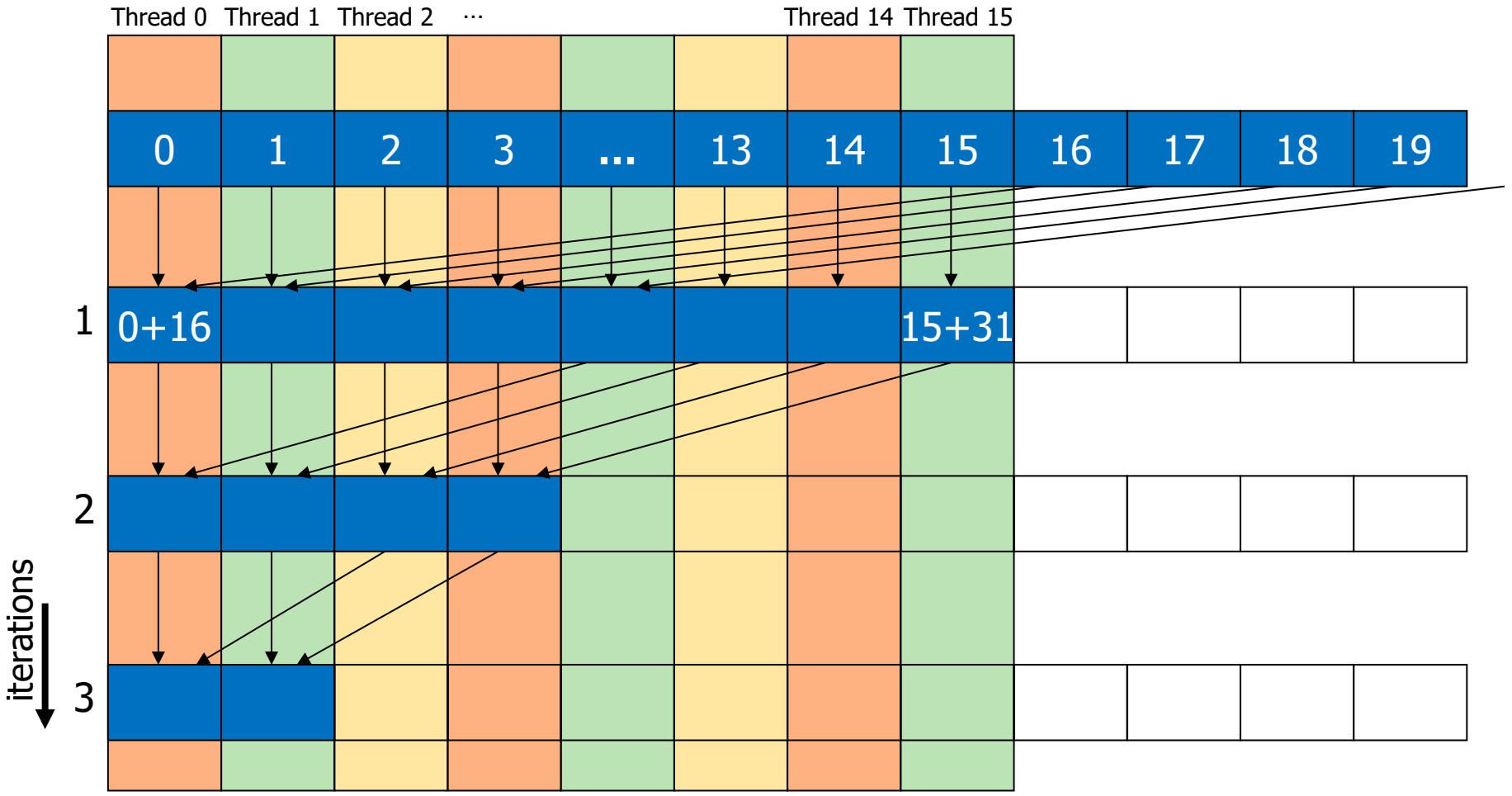
- A reduction operation reduces a set of values to a single value
  - Sum, Product, Minimum, Maximum are examples

- Properties of reduction
  - Associativity
  - Commutativity
  - Identity value

- Reduction is a key primitive for parallel computing
  - E.g., MapReduce programming model

Divergence-Free Mapping (I)

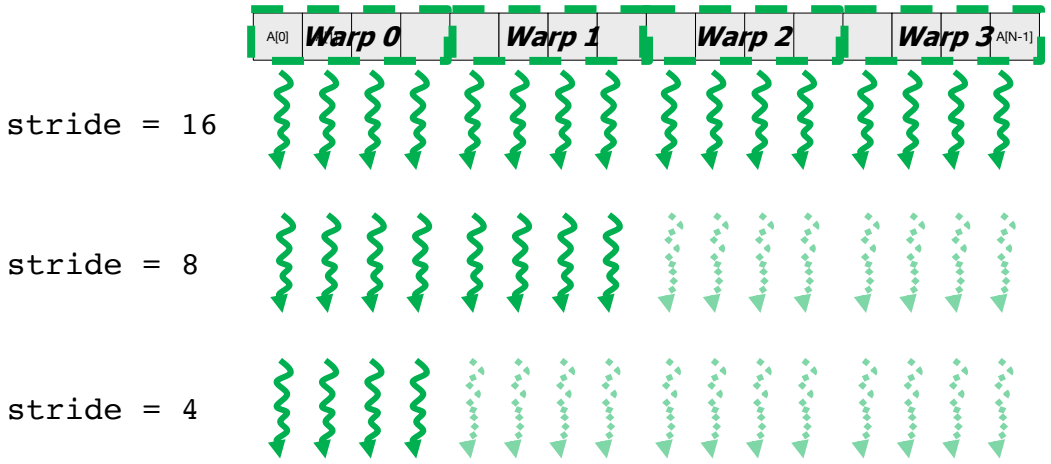
- All active threads belong to the same warp



Divergence-Free Mapping (II)

- Program with high SIMD utilization

```
__shared__ float partialSum[]

unsigned int t = threadIdx.x;

for(int stride = blockDim.x; stride > 0;  stride >> 1){

  __syncthreads();

  if (t < stride)
    partialSum[t] += partialSum[t + stride];

}
```



Warp utilization is maximized

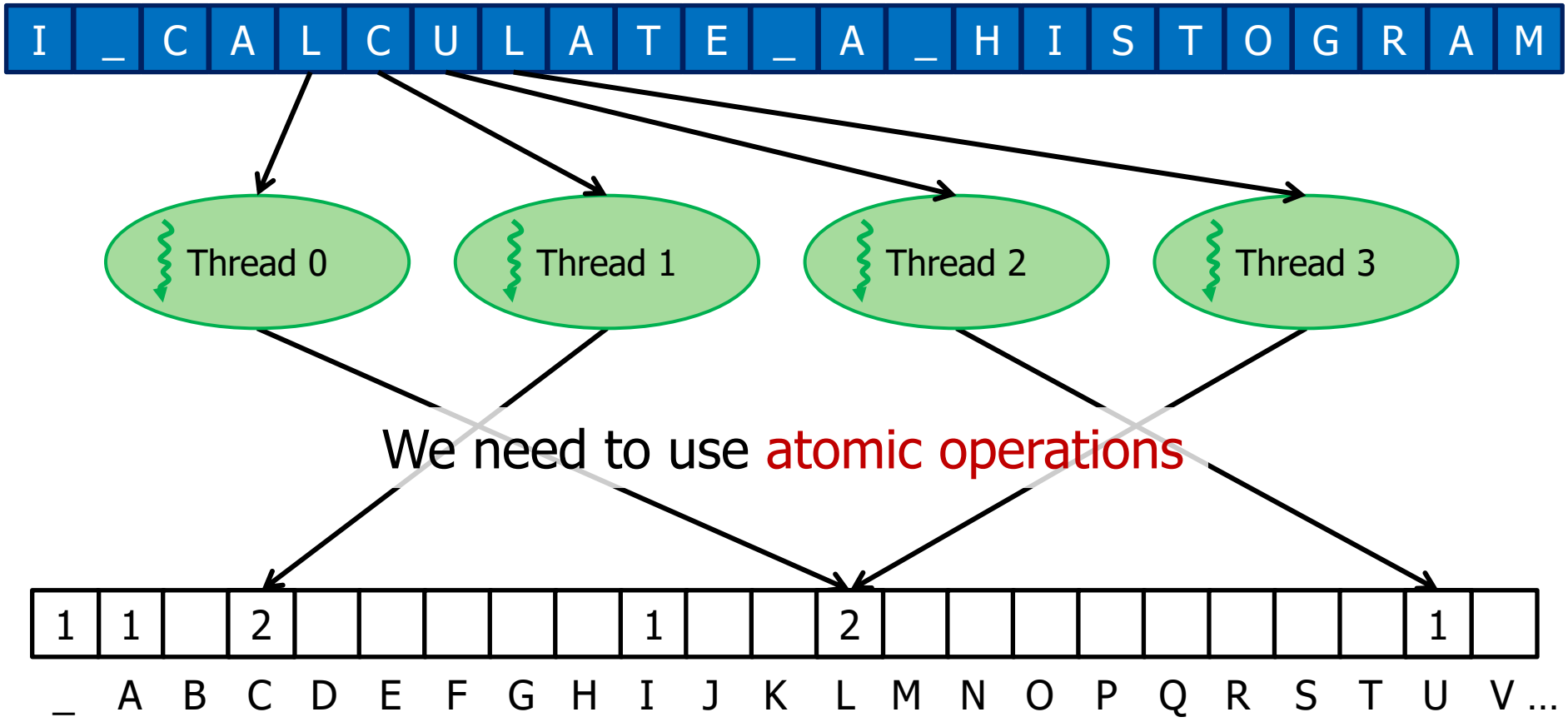
Histogram Computation

- Histogram is a frequently used computation for reducing the dimensionality and extracting notable features and patterns from large data sets
  - Feature extraction for object recognition in images
  - Fraud detection in credit card transactions
  - Correlating heavenly object movements in astrophysics
  - …
- Basic histograms - for each element in the data set, use the value to identify a "bin" to increment
  - Divide possible input value range into "bins"
  - Associate a counter to each bin
  - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

Slide credit: Hwu & Kirk

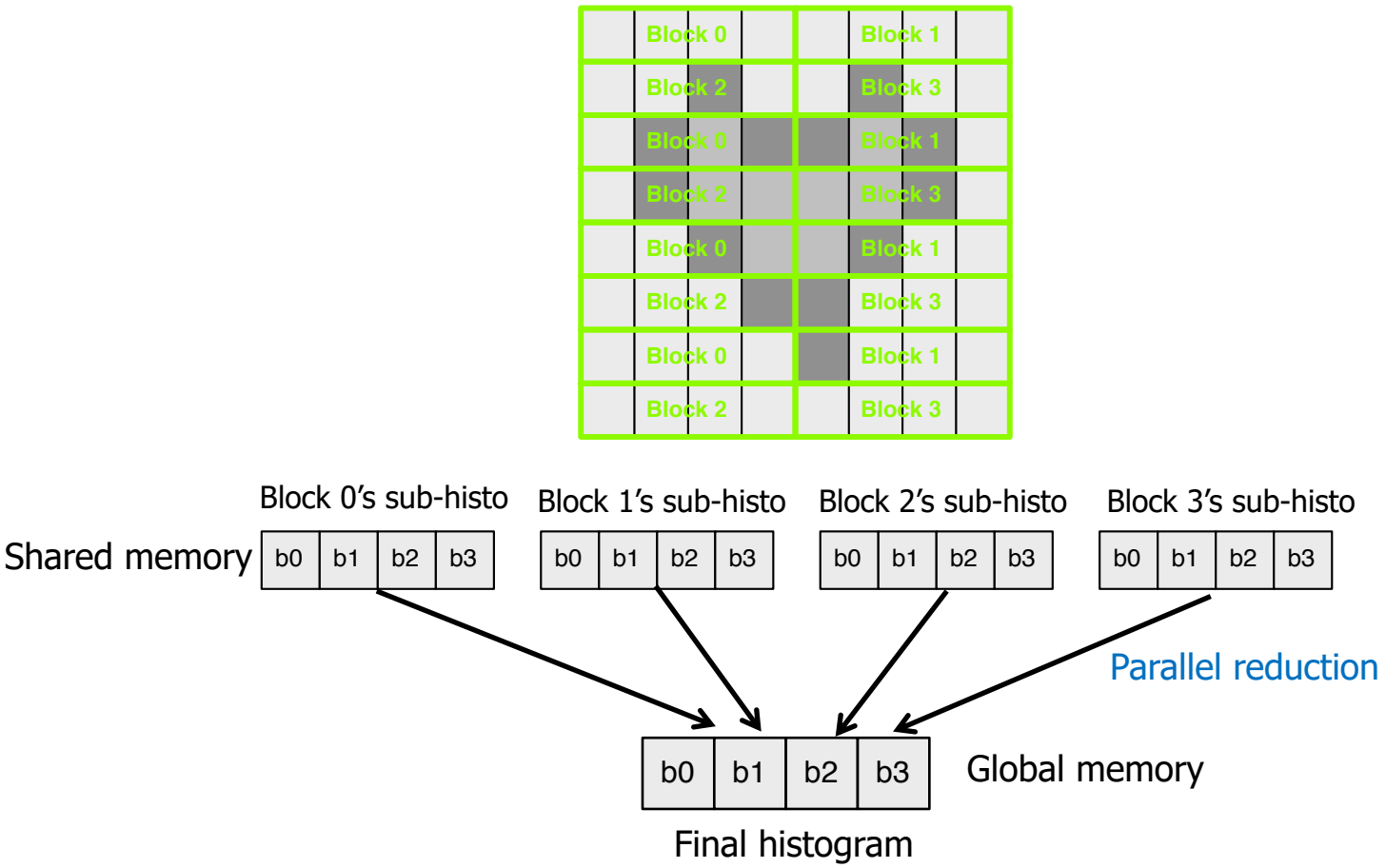
Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
  - Each thread moves to element threadID + #threads



Histogram Privatization

- Privatization: Per-block sub-histograms in shared memory
  - Threads use atomic operations in shared memory



Convolution Applications

- Convolution is a widely-used operation in signal processing, image processing, video processing, and computer vision

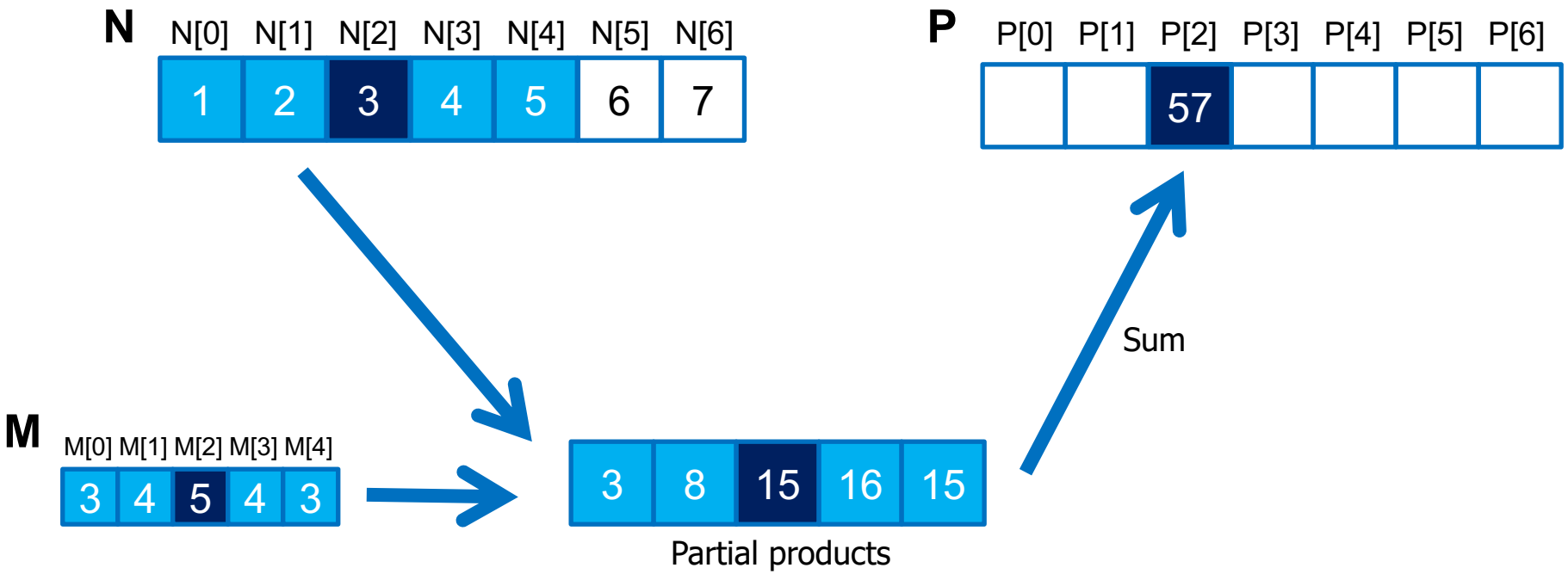
- Convolution applies a filter or mask or kernel* on each element of the input (e.g., a signal, an image, a frame) to obtain a new value, which is a weighted sum of a set of neighboring input elements
  - Smoothing, sharpening, or blurring an image
  - Finding edges in an image
  - Removing noise, etc.

- Applications in machine learning and artificial intelligence
  - Convolutional Neural Networks (CNN or ConvNets)

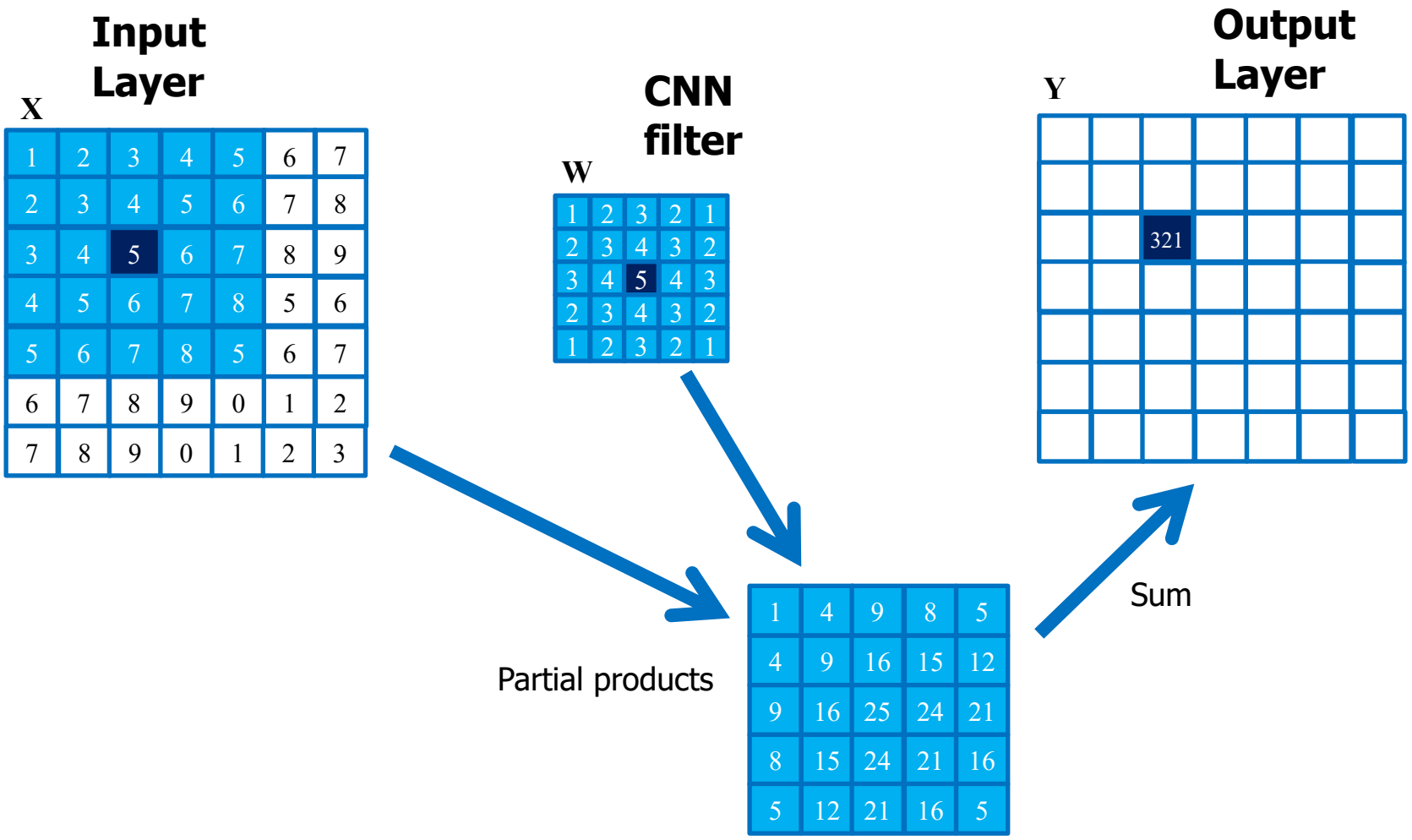
* The term "kernel" may create confusion in the context of GPUs (recall a CUDA/GPU kernel is a function executed by a GPU)

1D Convolution Example

- Commonly used for audio processing
- Mask size is usually an odd number of elements for symmetry (5 in this example)
- Calculation of P[2]:

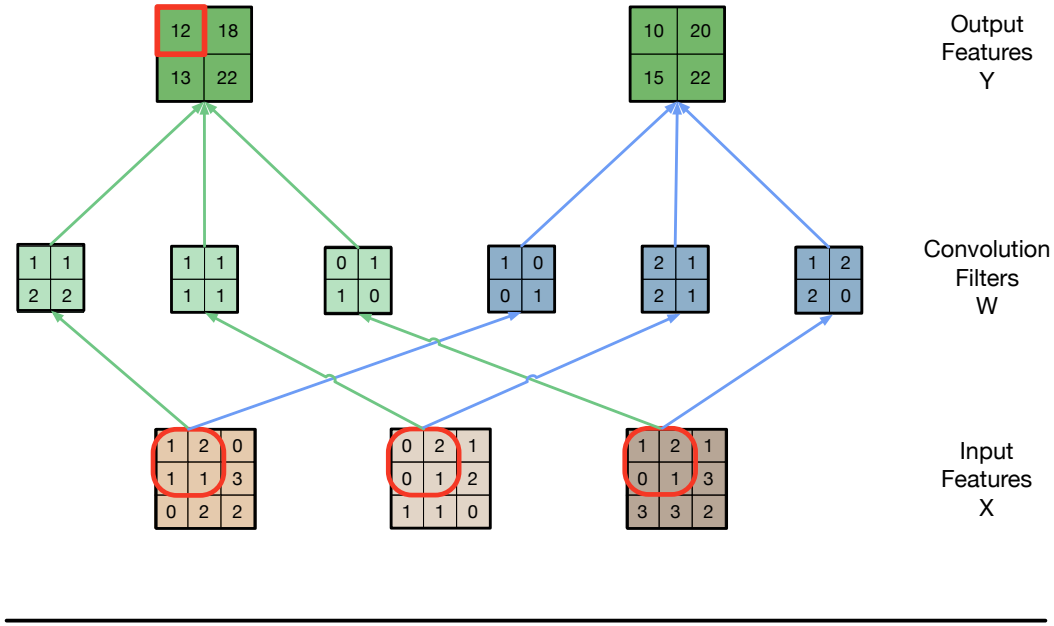


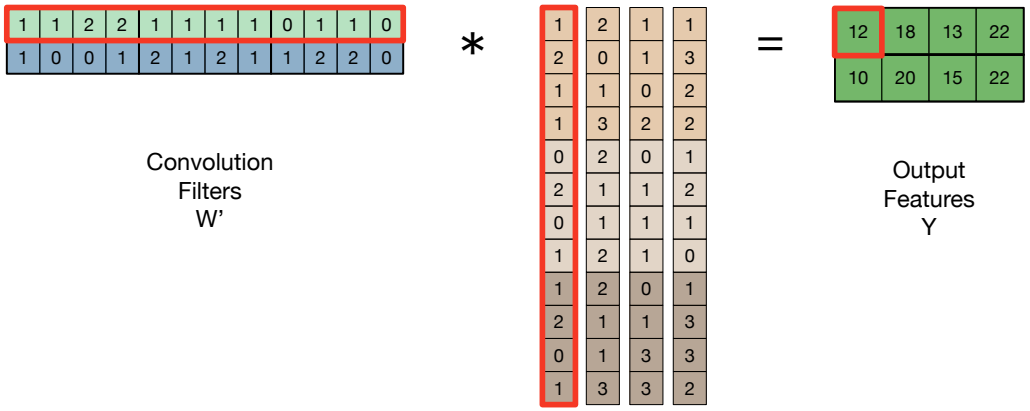
Another Example of 2D Convolution



Slide credit: Hwu & Kirk

Implementing a Convolutional Layer with Matrix Multiplication





Slide credit: Reproduced from Hwu & Kirk

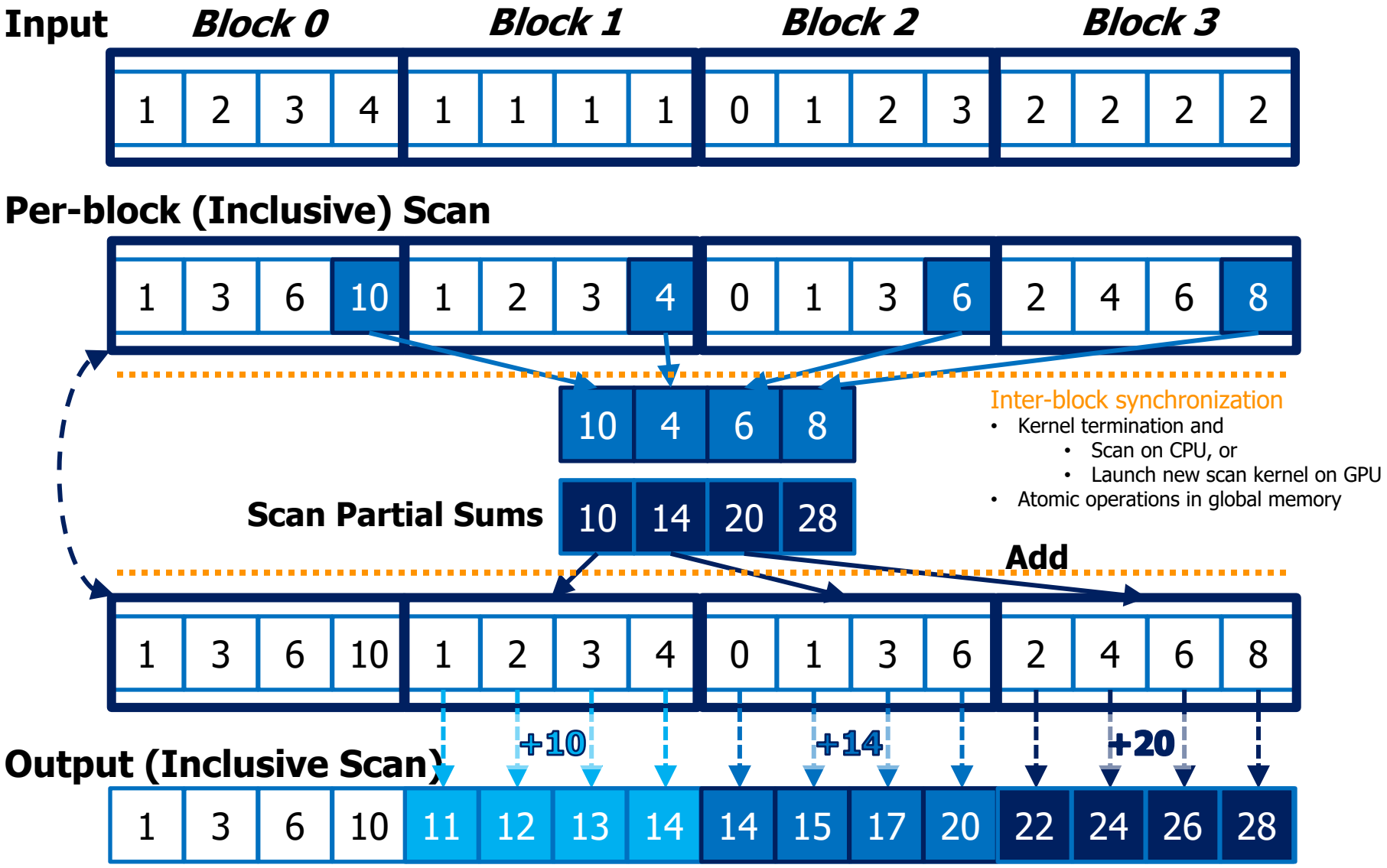
Prefix Sum (Scan)

- Prefix sum or scan is an operation that takes an input array and an associative operator,
  - E.g., addition, multiplication, maximum, minimum
- And returns an output array that is the result of recursively applying the associative operator on the elements of the input array

- Input array $[x_0, x_1, \ldots, x_{n-1}]$
- Associative operator $\oplus$

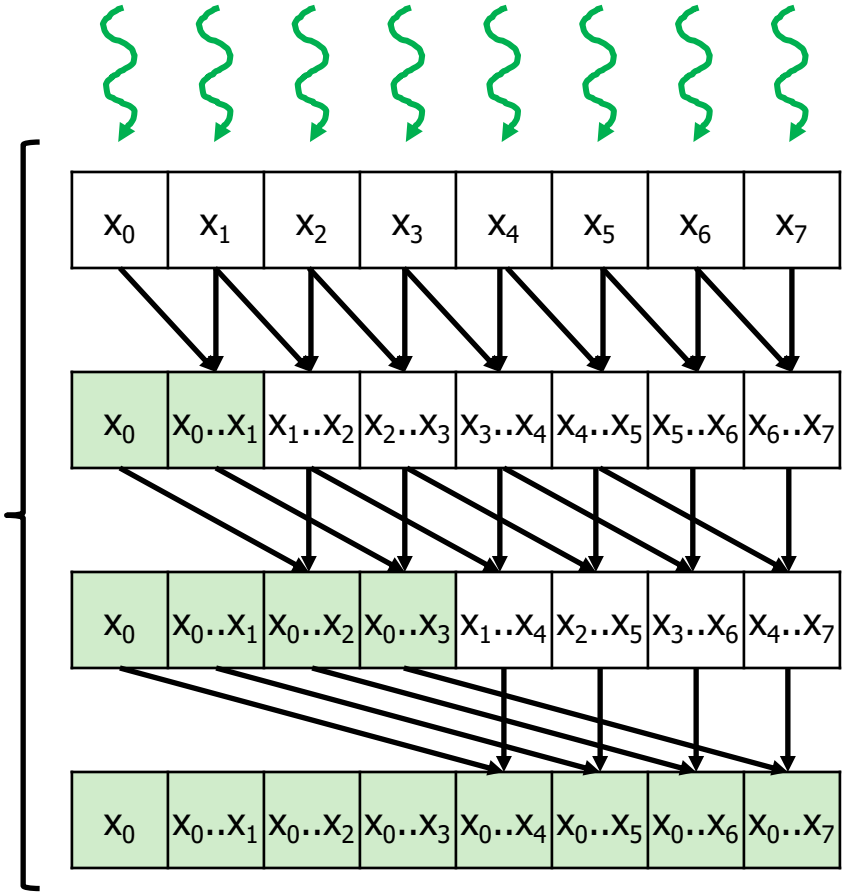
- An output array $[y_0, y_1, \ldots, y_{n-1}]$ where
  - Exclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_{i-1}$
  - Inclusive scan: $y_i = x_0 \oplus x_1 \oplus \ldots \oplus x_i$

Hierarchical (Inclusive) Scan



Kogge-Stone Parallel (Inclusive) Scan

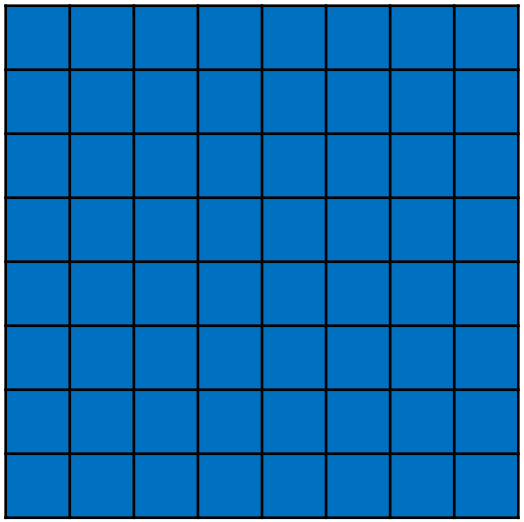
Observation:
memory locations are reused



Slide credit: Izzat El Hajj

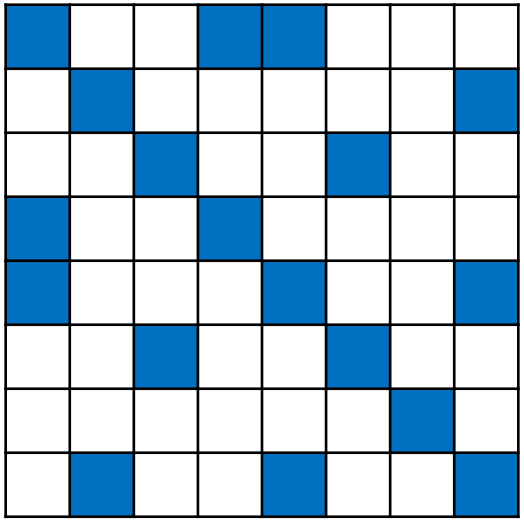
Sparse Matrices

A dense matrix is one where the majority of elements are not zero



A sparse matrix is one where many elements are zero

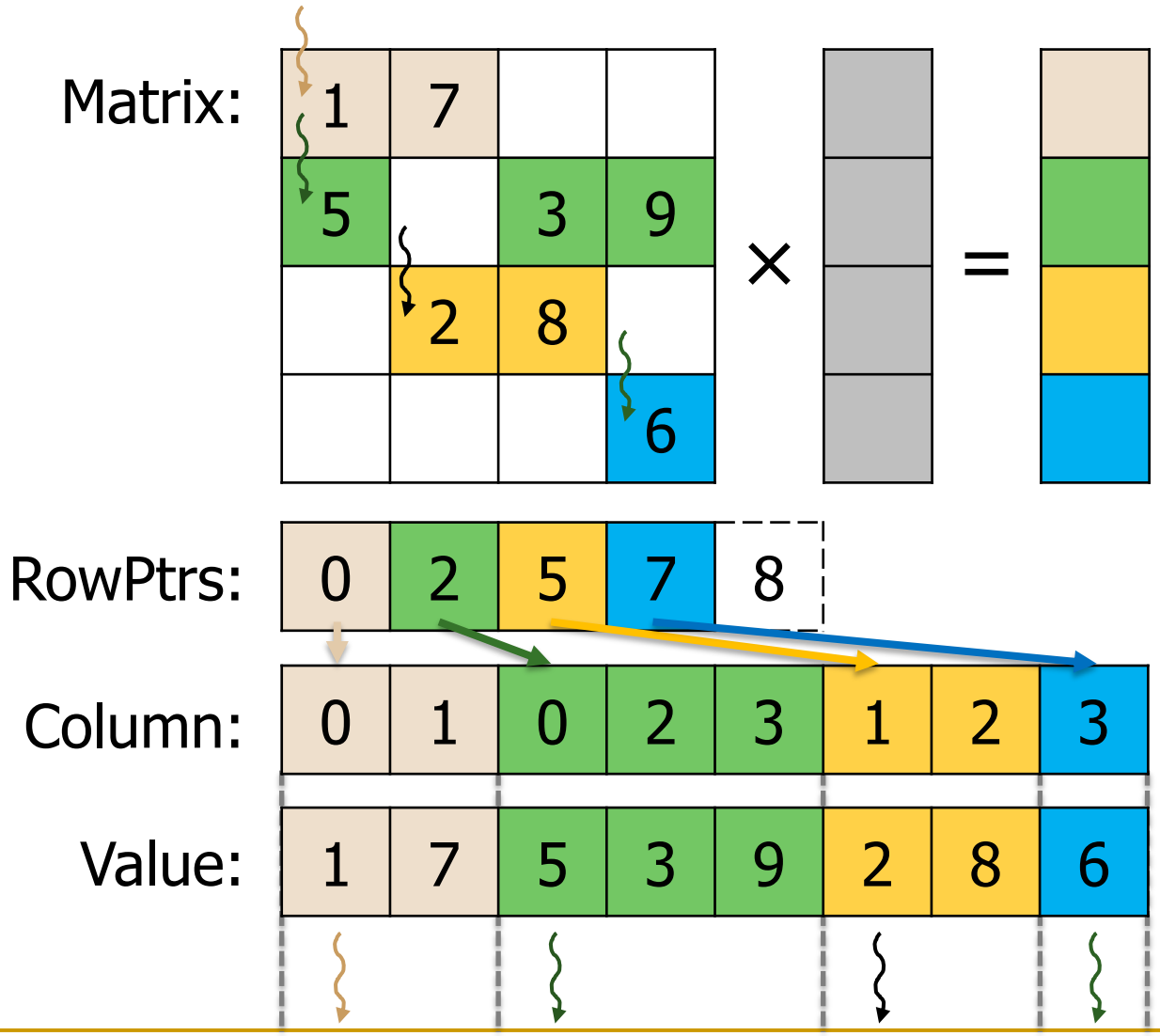
(many real world systems are sparse)



- Opportunities:
  - Do not need to allocate space for zeros (save memory capacity)
  - Do not need to load zeros (save memory bandwidth)
  - Do not need to compute with zeros (save computation time)

Slide credit: Izzat El Hajj

SpMV/CSR

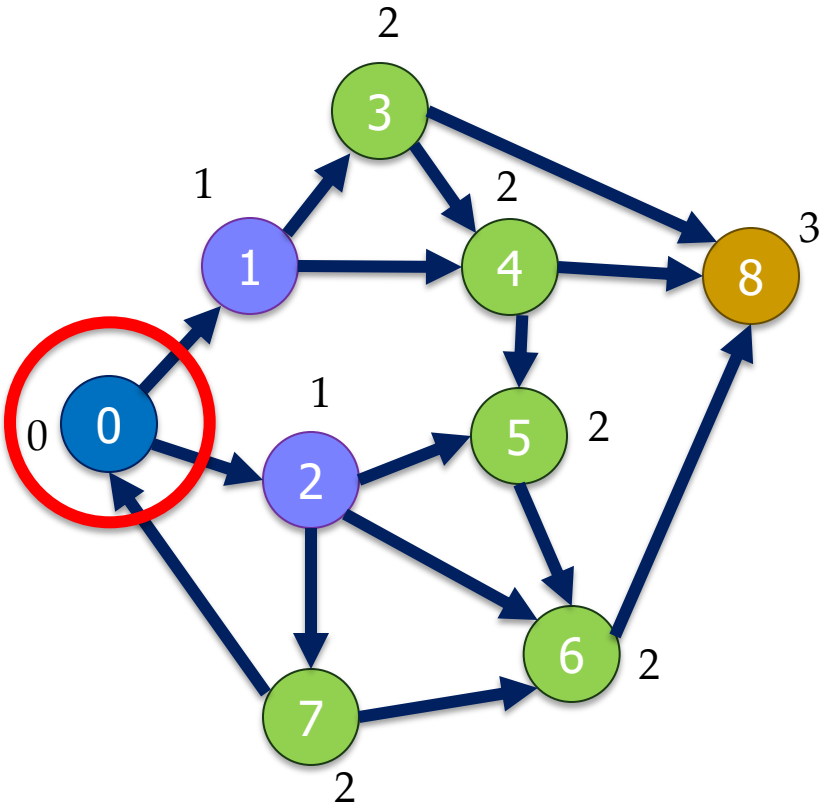


Parallelization approach:

Assign one thread to loop over each input row sequentially and update corresponding output element

Slide credit: Izzat El Hajj

Breadth-First Search – 3 Hops

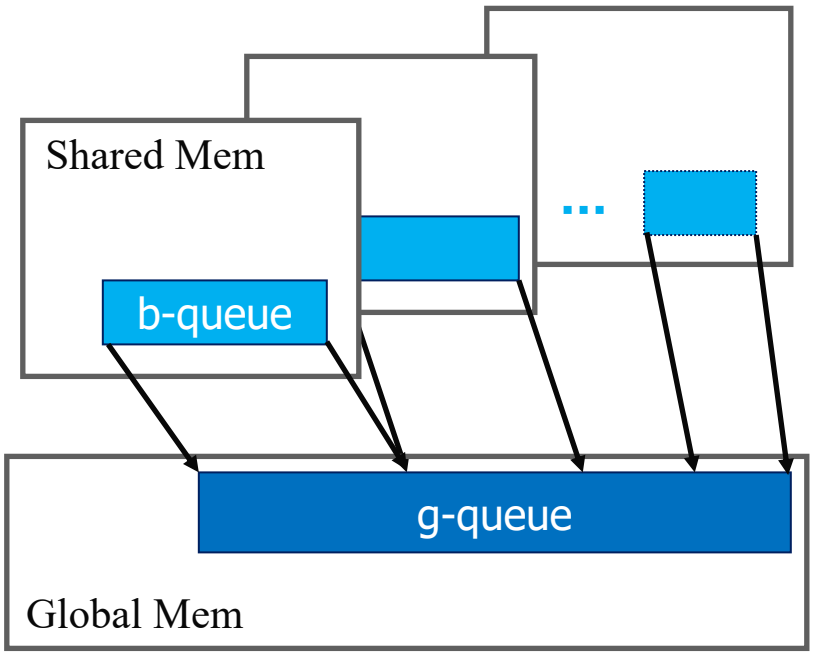


Desirable Outcome

- First Frontier (level 1 nodes)
  - 1, 2
- Second frontier (level 2 nodes)
  - 3, 4, 5, 6, 7
- Third frontier (level 3 nodes)
  - 8
- …

Slide credit: Hwu & Kirk

Two-level Hierarchy

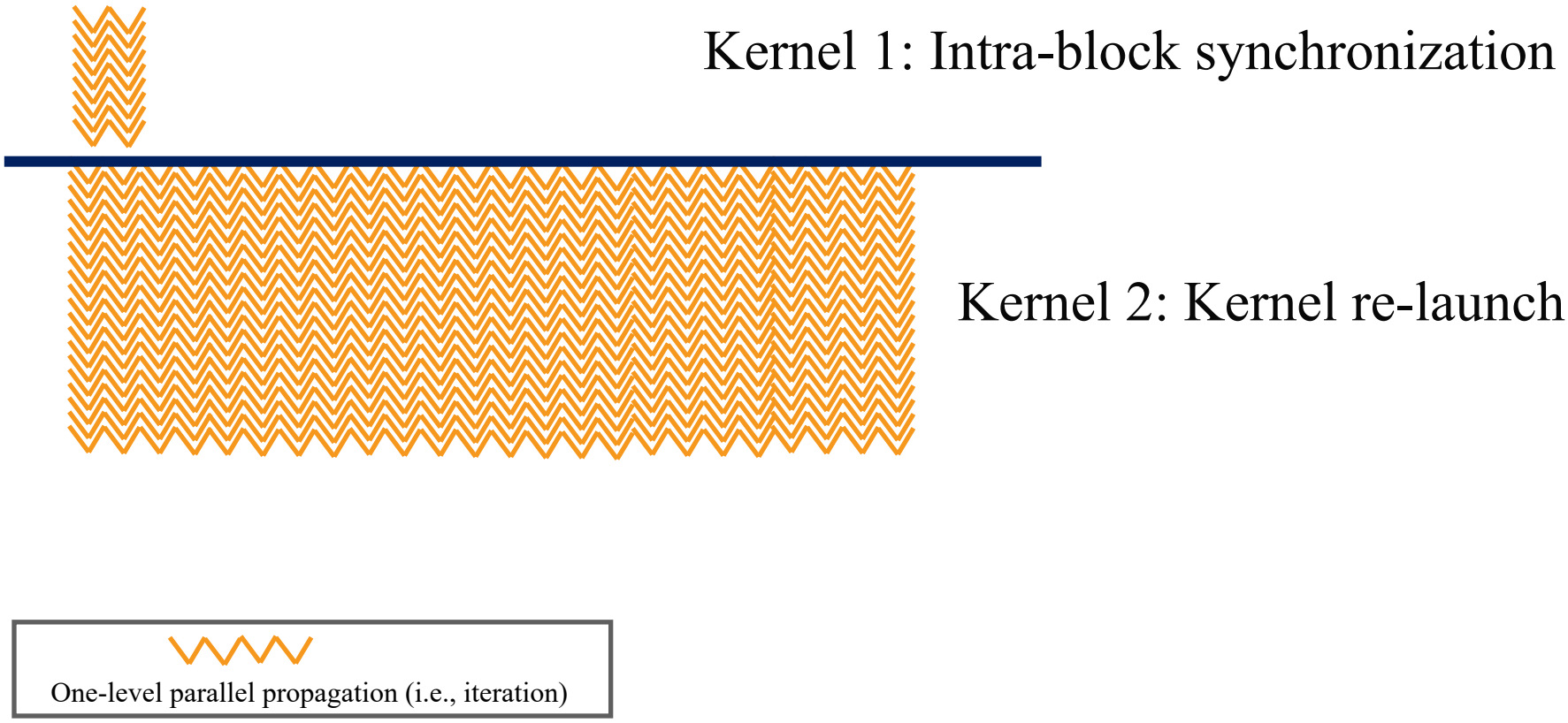


- Block queue (b-queue)
  - Inserted by all threads in a block
  - Resides in Shared Memory
- Global queue (g-queue)
  - Inserted only when a block completes
- Problem:
  - Collision on b-queues
  - Threads in the same block can cause heavy contention

Slide credit: Hwu & Kirk

Hierarchical Kernel Arrangement

- Customize kernels based on the size of frontiers
- Use fast barrier synchronization when the frontier is small



Slide credit: Hwu & Kirk

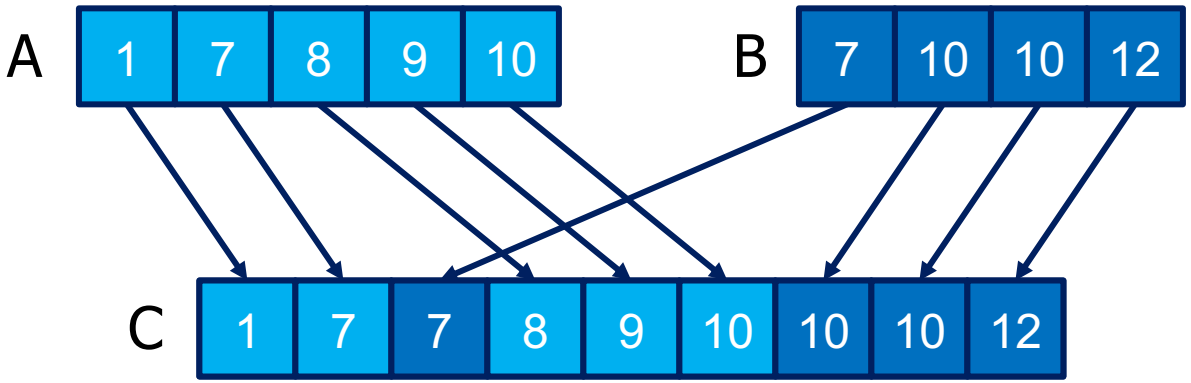
Merge Sort

Merge Sort

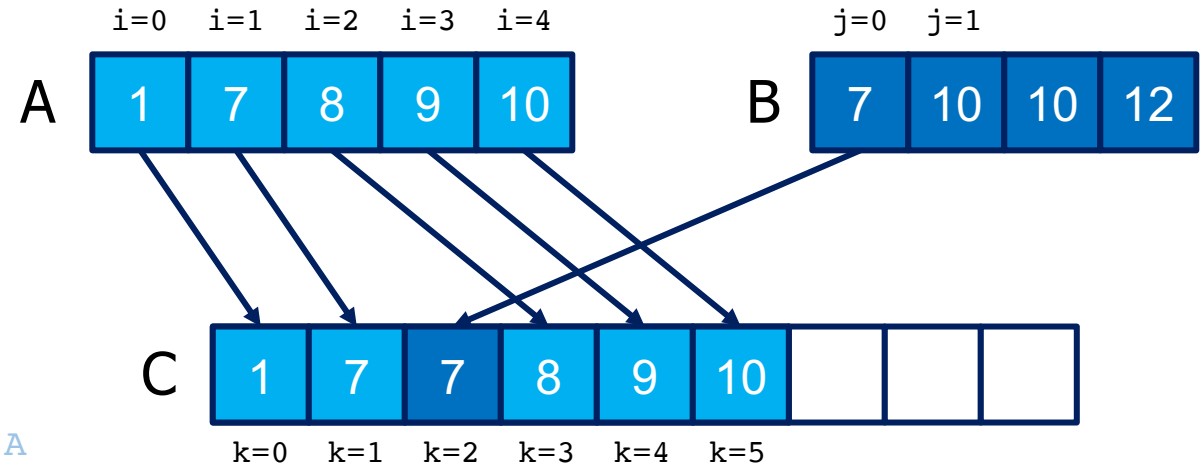
- Merge operation takes two sorted lists as inputs and generates one combined, sorted list
  - There is an order relation (e.g., *less than or equal to*)
  - It may be stable (i.e., maintain the relative order of elements with the same value/key)
  - and may have preference (e.g., elements of list A go before elements of list B if they have the same value)
- Merge is an important building block of sorting algorithms
- It is also frequently used in the reduce step of MapReduce frameworks
- The dynamic nature of data accesses makes it challenging to exploit locality for memory access efficiency

Merging Two Sorted Lists into One

- There is an order relation (e.g., less than or equal to) in the sorted lists and in the merged list
- We focus on stable sort
- Whenever A and B have elements of the same value, the element from A goes first

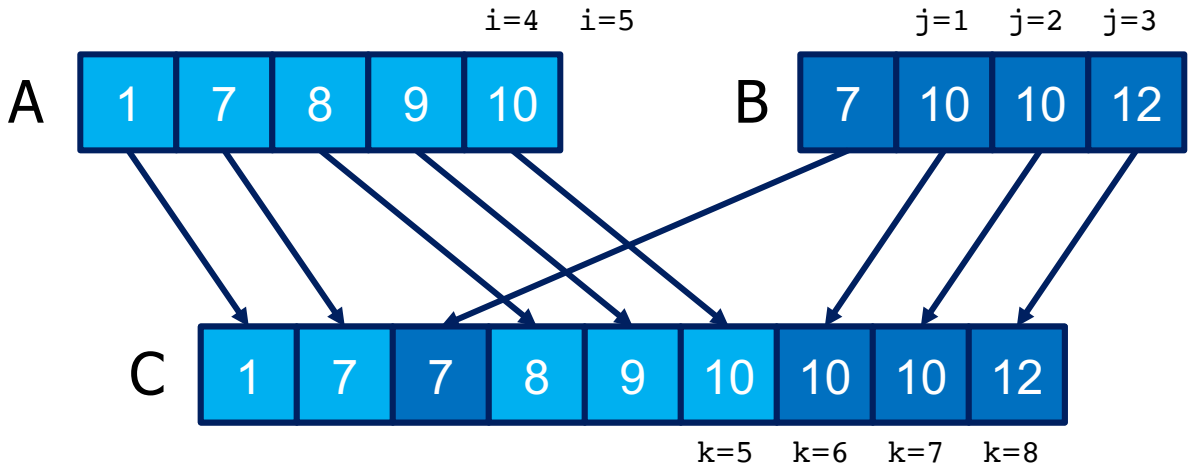


A Sequential Merge (I)




```
void merge_sequential(
          int *A, int m,
          int *B, int n,
          int *C) {
  int i = 0; // Index into A
  int j = 0; // Index into B
  int k = 0; // Index into C

  // Handle the comparison of A[] and B[]
  while ((i < m) && (j < n)) {
    if (A[i] <= B[j]) {
      C[k++] = A[i++];
    } else {
      C[k++] = B[j++];
    }
  }
...
```

A Sequential Merge (II)

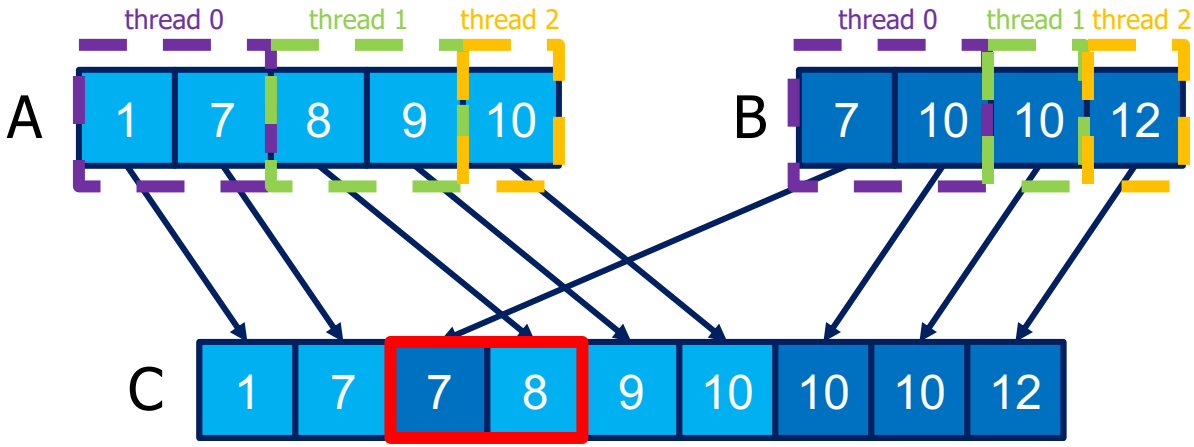



```
...
  if (i == m) {
  // Done with A[] handle remaining B[]
    for (; j < n; j++) {
      C[k++] = B[j];
    }
  } else {
  // Done with B[], handle remaining A[]
    for (; i < m; i++) {
      C[k++] = A[i];
    }
  }
}
```

Sequential algorithm complexity
O(m + n)

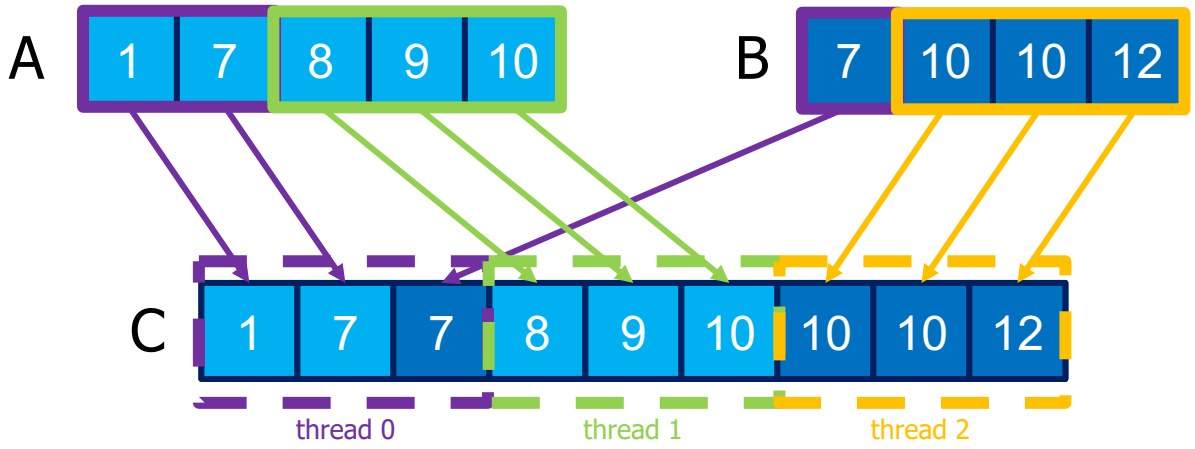
Parallel Merge

- Scatter parallelization?
  - Each thread would take a section of list A and a section of list B, and would find the element locations in list C
  - The destination of an element of A or an element of B depends on the elements of the other list
  - This includes the elements of A and B assigned to other threads



Parallel Merge: Gather Parallelization (I)

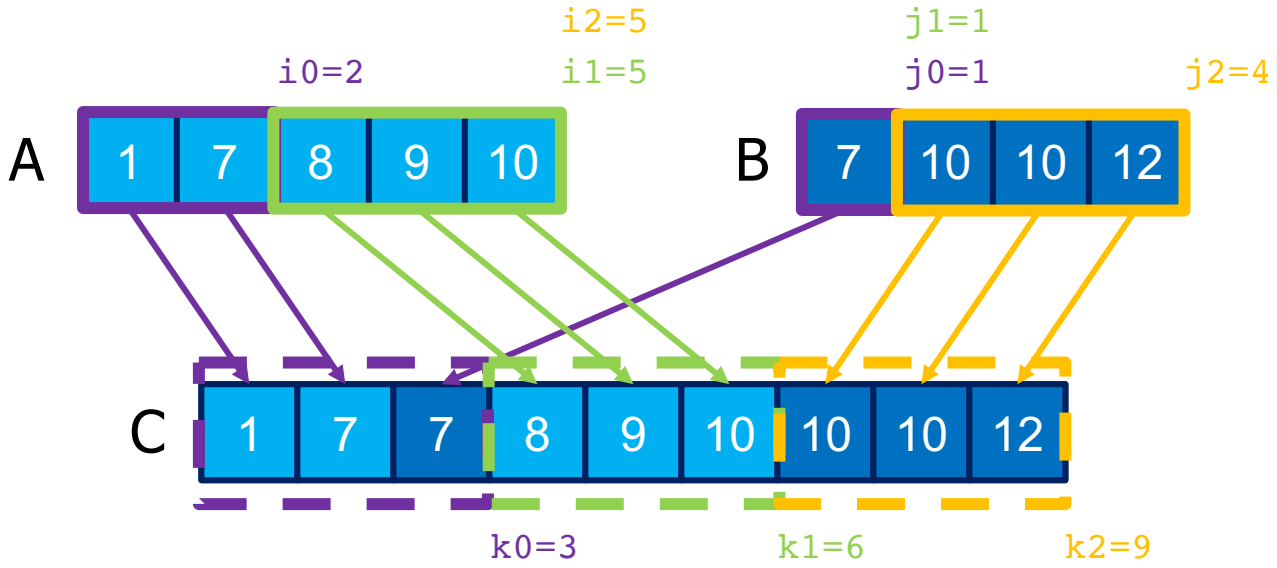
- We partition the output list equally among threads
- Each thread collects input elements for its section of the output



The range of input elements to be used by each thread is a function of the input elements

Parallel Merge: Gather Parallelization (II)

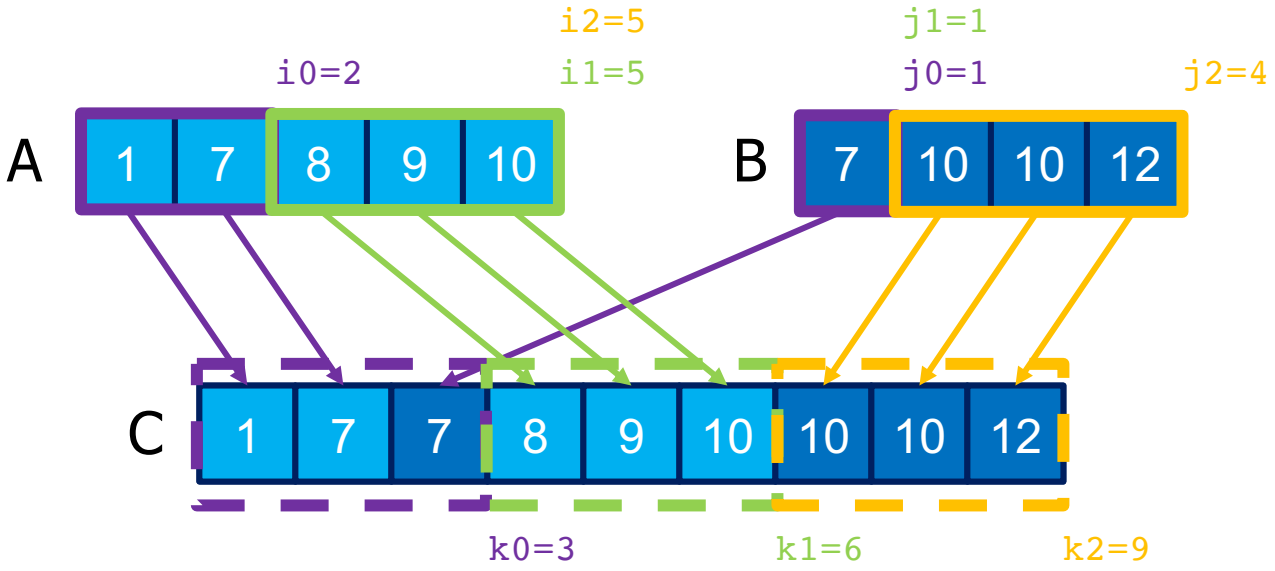
- Observation
  - For any `k` such that `0≤k<m+n`, we can find `i` and `j` such that `k=i+j`, `0≤i<m` and `0≤j<n`



For an element `C[k]`, `k` is referred to as its rank and `i` and `j` are referred to as its co-ranks

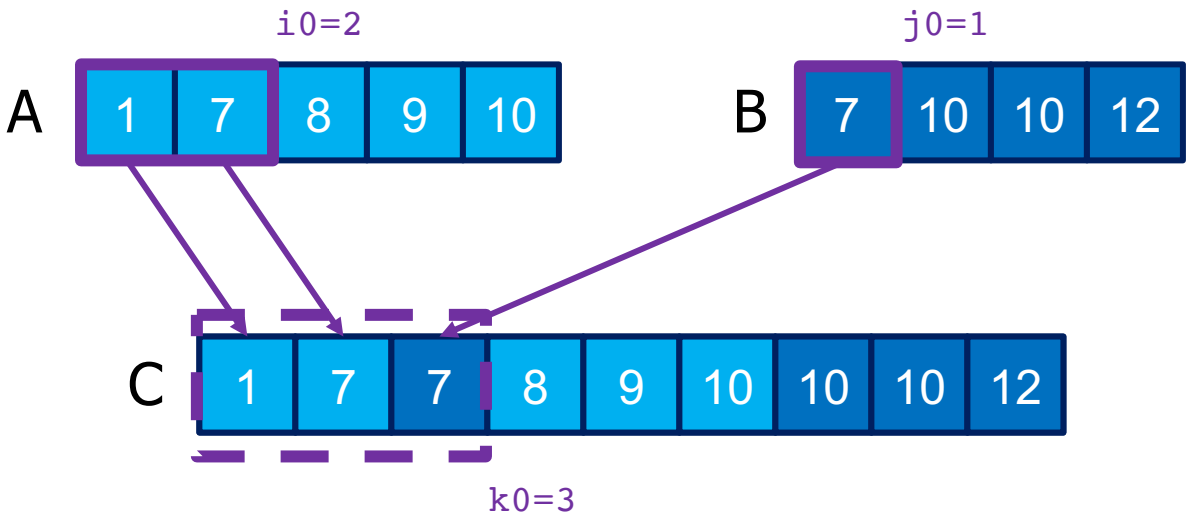
Parallel Merge: Gather Parallelization (III)

- Each thread gathers the elements of a continuous section of the output
- All threads identify the starting and ending locations of the continuous sections of the inputs (A and B) that they will use
  - Input identification and partition by finding co-ranks
- All threads perform merge for their sections in parallel
  - Each thread executes a sequential merge for its own section



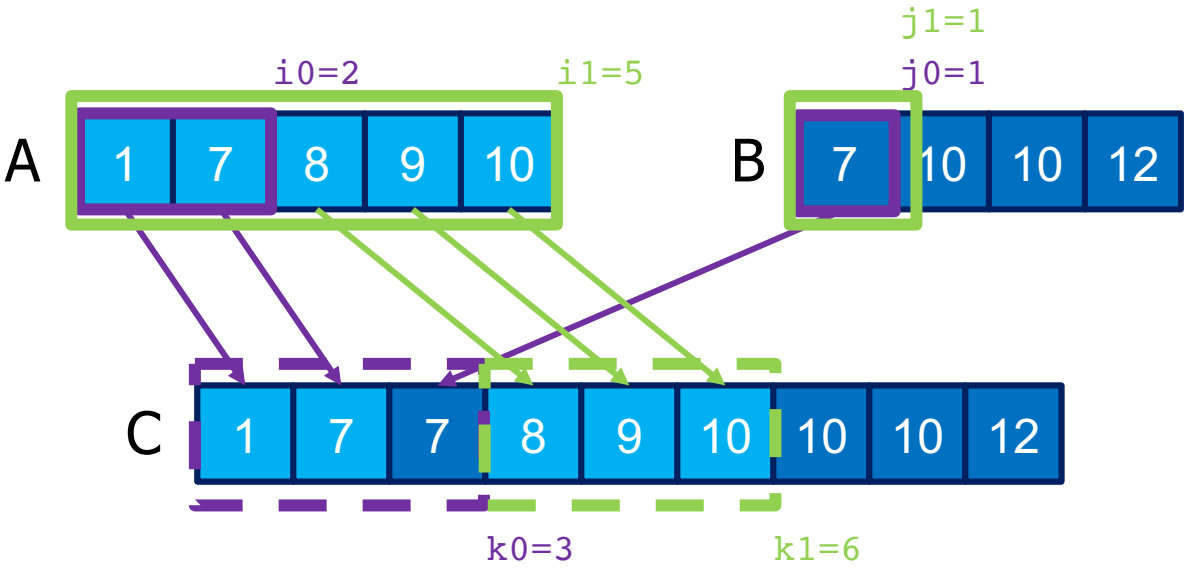
Co-Rank Value Calculation (I)

- The co-rank values of a prefix string leading to the `k`th output element is a pair of `i` and `j` values that specify the rank values of the prefix strings of A and B that are used in forming the output prefix string leading to that `k`th output element
- For a given output prefix string of rank `k`, its co-rank values can be found by searching for `i` and `j` such that `k=i+j` and
  - `A[i-1] ≤ B[j]`
  - `B[j-1] < A[i]`



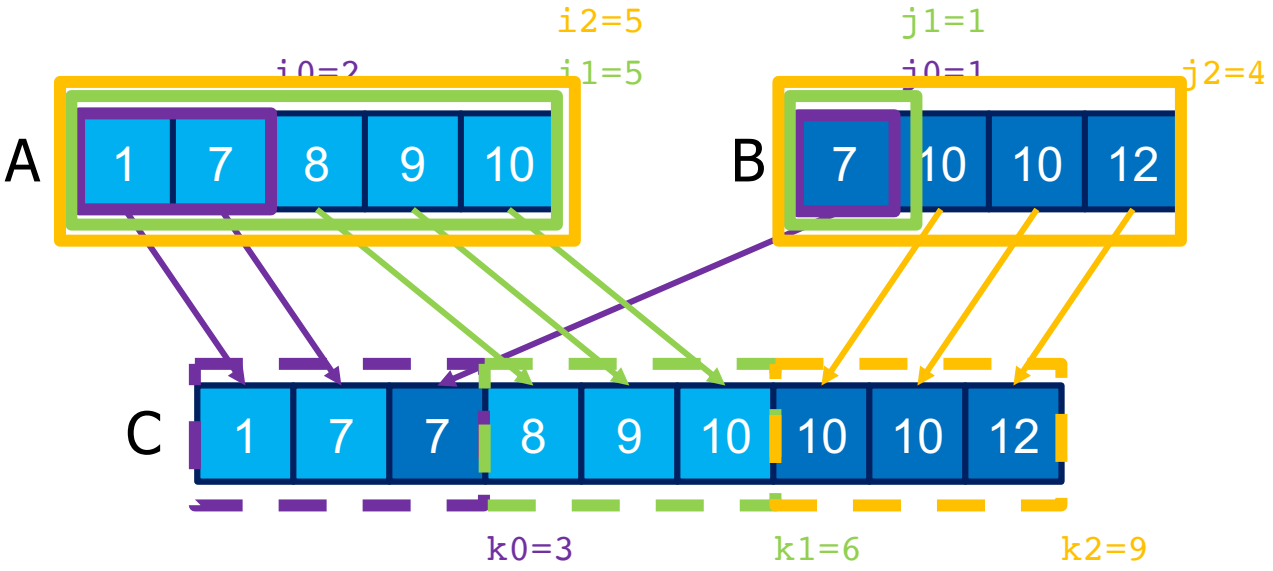
Co-Rank Value Calculation (II)

- Finding co-rank values for different threads is not balanced
- The search range for a prefix string is a subset of that of a higher ranked one
  - Finding `A[2] ≤ B[1]` and `B[0] < A[3]` needs a smaller search range than finding `A[4] ≤ B[1]` and `B[0] < A[4]`



Co-Rank Function (I)

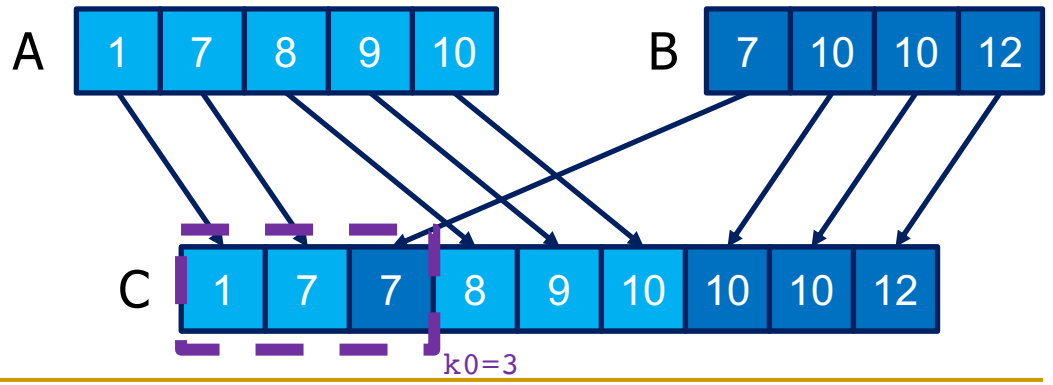
- Use binary search (or n-ary search) for `i` and `j` values to minimize the effect of increasing search ranges
  - Reduces the computational complexity from O(N) to O(log N)
- Co-rank function
  - `int co_rank(int k, int* A, int m, int* B, int n)`
  - Returns `i`, and `j` is derived as `k – i`



Co-Rank Function (II)

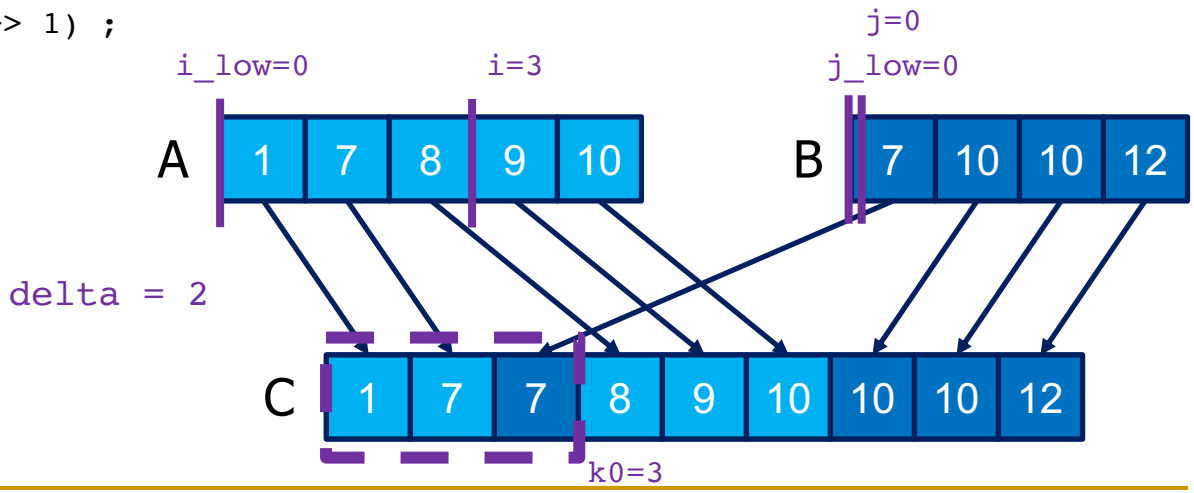
```
int co_rank(int k, int* A, int m, int* B, int n) {
  int i = k<m ? k : m;  // i = min(k, m)
  int j = k-i;
  int i_low = 0>(k-n) ? 0 : k-n;  // i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0 : k-m;  // j_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active)    {
    if (i > 0 && j < n && A[i-1] > B[j]) {
      delta = ((i - i_low +1) >> 1) ;  // ceil(i-i_low)/2)
      j_low = j;
      j = j + delta;
      i = i - delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
      delta = ((j - j_low +1) >> 1) ;
      i_low = i;
      i = i + delta;
      j = j - delta;
    } else {
      active = false;
    }
  }
  return i;
}
```

```
int i = co_rank(3, A, 5, B, 4);
```



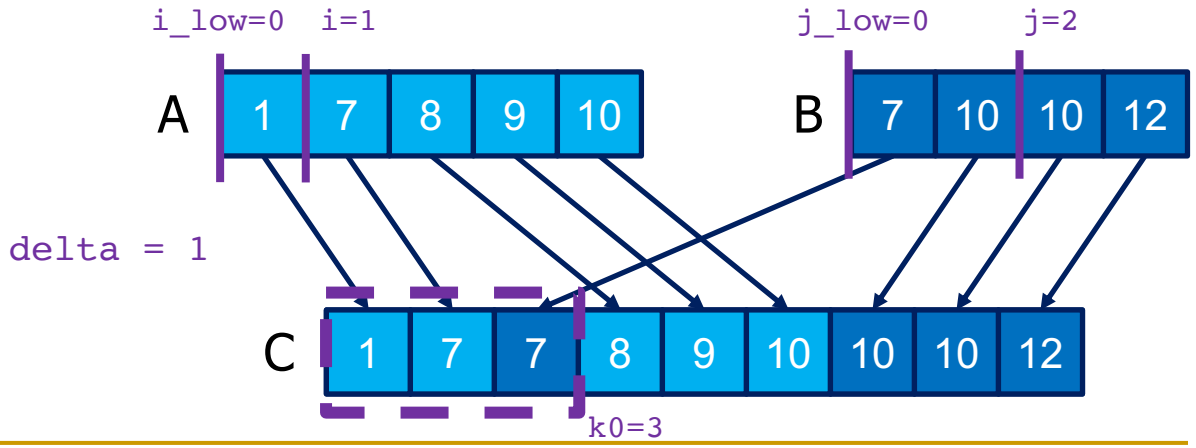
Co-Rank Function: Iteration 0

```
int co_rank(int k, int* A, int m, int* B, int n) {
  int i = k<m ? k : m;  // i = min(k, m)
  int j = k-i;
  int i_low = 0>(k-n) ? 0 : k-n;  // i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0 : k-m;  // j_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active)    {
    if (i > 0 && j < n && A[i-1] > B[j]) {
      delta = ((i - i_low +1) >> 1) ;  // ceil(i-i_low)/2)
      j_low = j;
      j = j + delta;
      i = i - delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
      delta = ((j - j_low +1) >> 1) ;
      i_low = i;
      i = i + delta;
      j = j - delta;
    } else {
      active = false;
    }
  }
  return i;
}
```



Co-Rank Function: Iteration 1

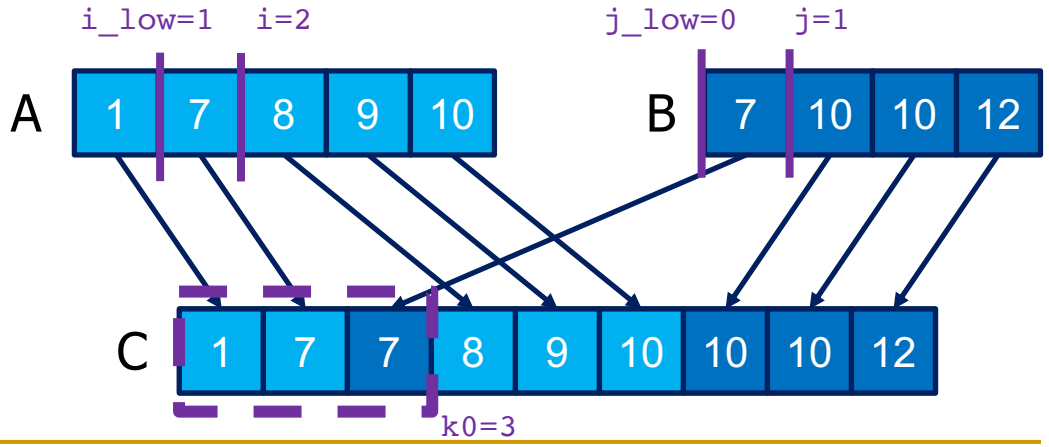
```
int co_rank(int k, int* A, int m, int* B, int n) {
  int i = k<m ? k : m;  // i = min(k, m)
  int j = k-i;
  int i_low = 0>(k-n) ? 0 : k-n;  // i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0 : k-m;  // j_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active)    {
    if (i > 0 && j < n && A[i-1] > B[j]) {
      delta = ((i - i_low +1) >> 1) ;  // ceil(i-i_low)/2)
      j_low = j;
      j = j + delta;
      i = i - delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
      delta = ((j - j_low +1) >> 1) ;
      i_low = i;
      i = i + delta;
      j = j - delta;
    } else {
      active = false;
    }
  }
  return i;
}
```



Co-Rank Function: Iteration 2

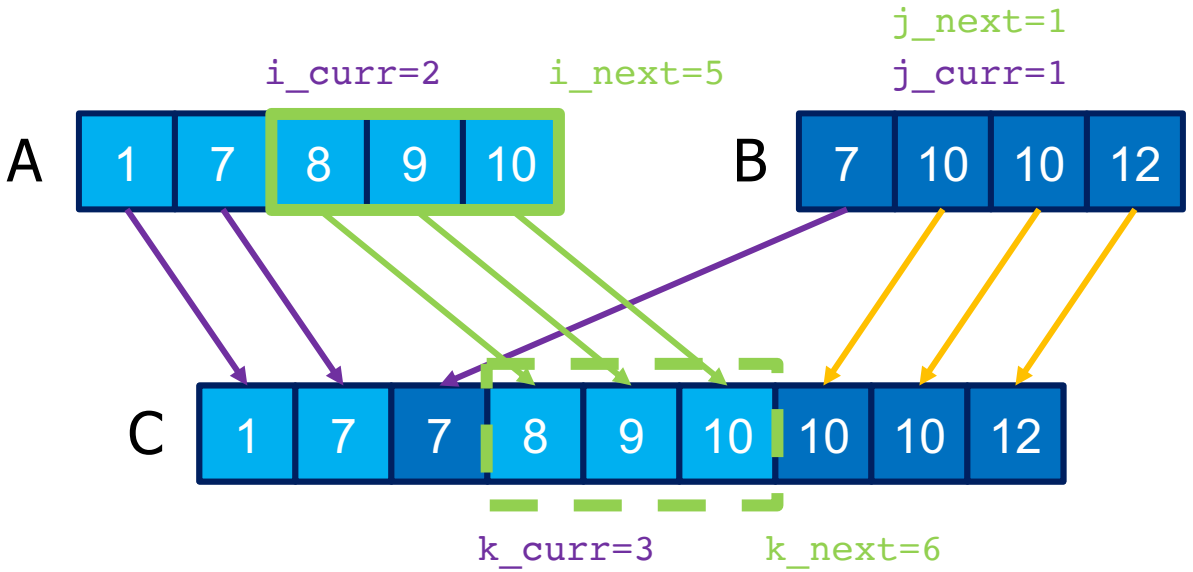
```
int co_rank(int k, int* A, int m, int* B, int n) {
  int i = k<m ? k : m;  // i = min(k, m)
  int j = k-i;
  int i_low = 0>(k-n) ? 0 : k-n;  // i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0 : k-m;  // j_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active)    {
    if (i > 0 && j < n && A[i-1] > B[j]) {
      delta = ((i - i_low +1) >> 1) ;  // ceil(i-i_low)/2)
      j_low = j;
      j = j + delta;
      i = i - delta;
    } else if (j > 0 && i < m && B[j-1] >= A[i]) {
      delta = ((j - j_low +1) >> 1) ;
      i_low = i;
      i = i + delta;
      j = j - delta;
    } else {
      active = false;
    }
  }
  return i;
}
```

i = 2



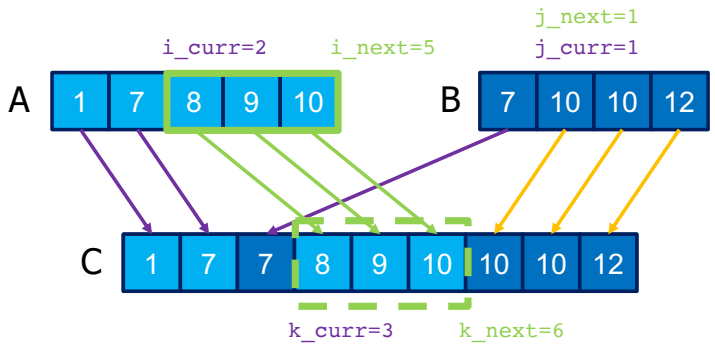
Basic Parallel Merge (I)

- Each thread is in charge of a continuous section of the output
  - `k_curr` and `k_next` define the output range for a thread `C[k_curr]` to `C[k_next]`
- Each thread calls the co-rank function to get `i_curr`, `i_next`, `j_curr`, and `j_next` values



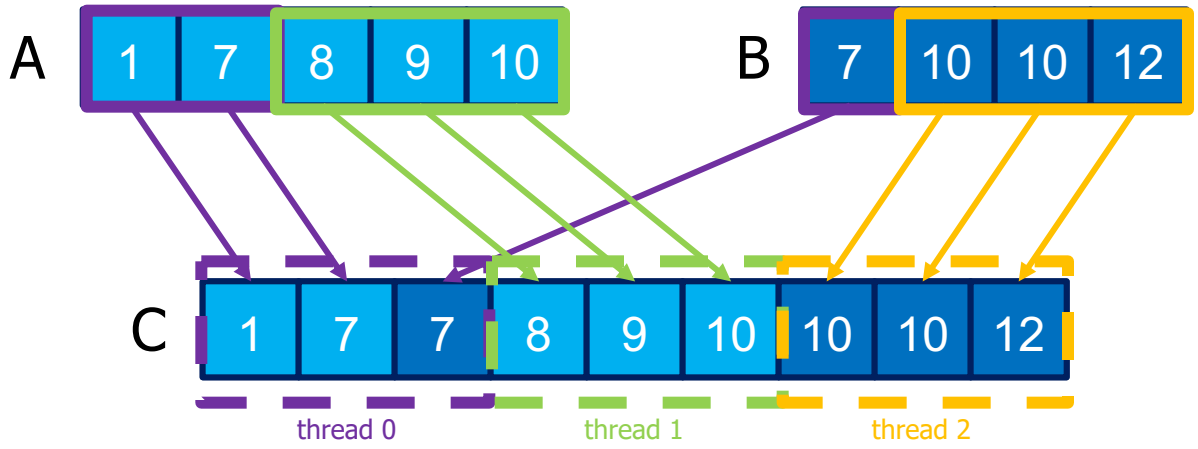
Basic Parallel Merge (II)

```
__global__ void merge_basic_kernel(int* A, int m, int* B, int n, int* C){
  int tid = blockIdx.x*blockDim.x + threadIdx.x;

  // Start index of output section
  int k_curr = tid * ceil((m+n)/(blockDim.x*gridDim.x);
  // End index of output section
  int k_next = min((tid+1) * ceil((m+n)/(blockDim.x*gridDim.x)), m+n);

  // Co-rank values
  int i_curr = co_rank(k_curr, A, m, B, n);
  int i_next = co_rank(k_next, A, m, B, n);

  int j_curr = k_curr - i_curr;
  int j_next = k_next - i_next;

  merge_sequential(&A[i_curr], i_next-i_curr, &B[j_curr], j_next-j_curr, &C[k_curr]);
}
```

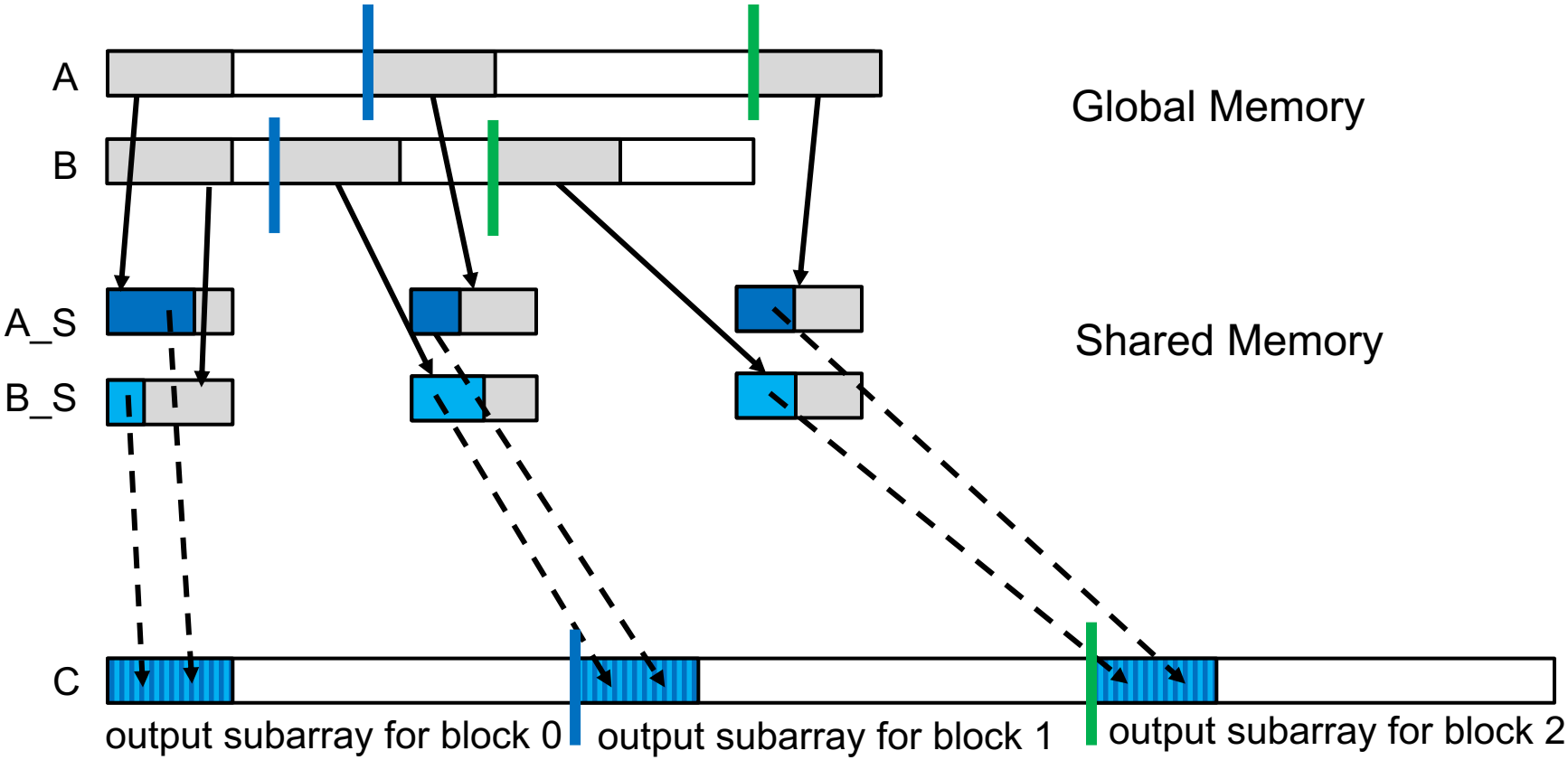


Basic Parallel Merge (III)

- Problem: Uncoalesced memory accesses
  - E.g., first read of thread 0 to A[0], thread 1 to A[2], thread 2 to B[1]; first write of thread 0 to C[0], thread 1 to C[3], thread 2 to C[6]
- Solution: Collaborative loading of sections of A and B into the shared memory – tiled merge kernel



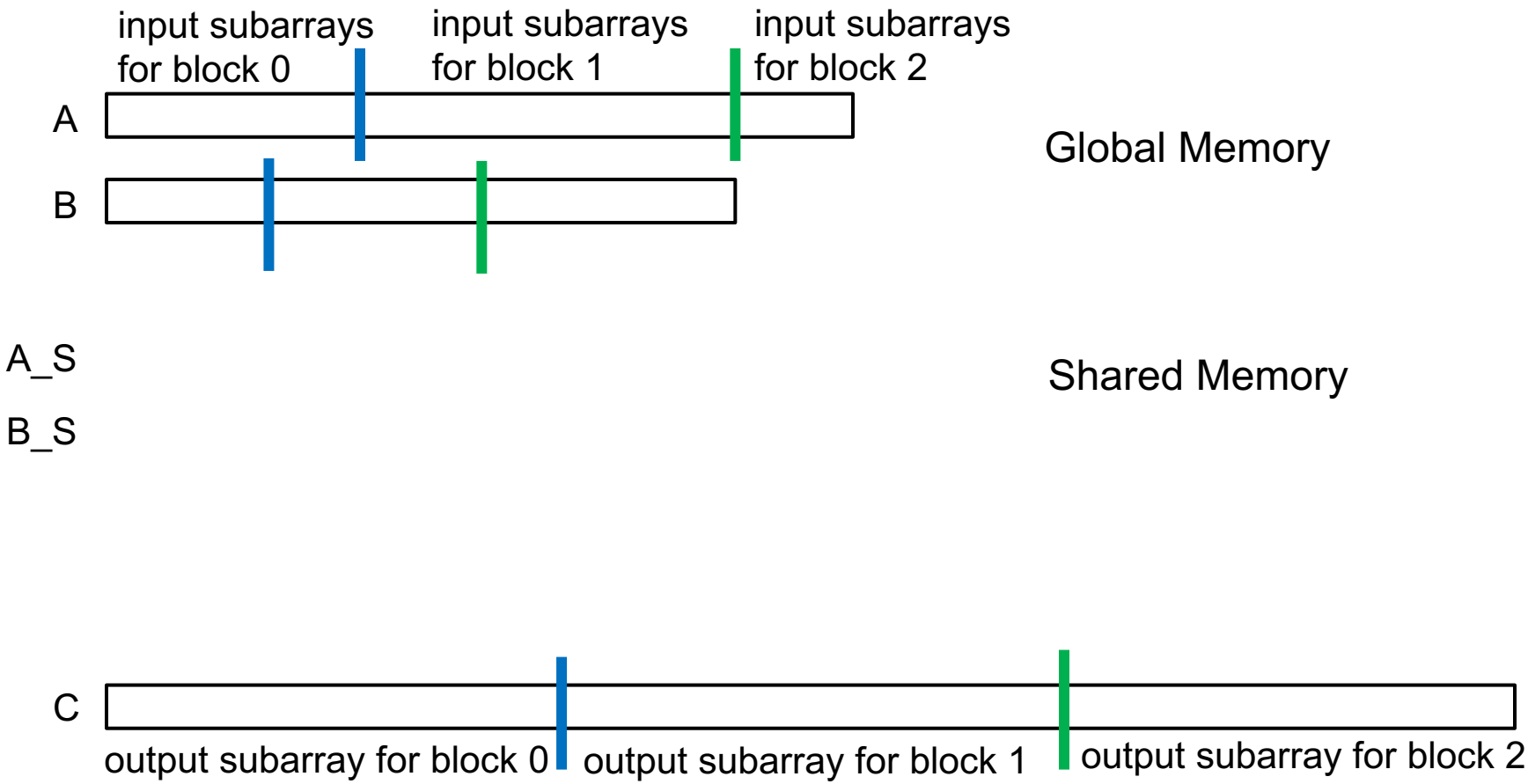
Tiled Merge Kernel (I)



Tiled Merge Kernel (II)

- Assign output sections to thread blocks
  - Each section should have at least a few thousand elements
- Leader thread of each block performs binary (n-ary) search to identify input sections
- Each block iteratively generates its output section
- In each iteration:
  - Threads of a block collaboratively load a tile of A and a tile of B into shared memory
    - Each tile should have at least a few hundred elements
  - Divide the output tile into subsections and assign them to threads
    - Each subsection should have ten or more of elements
  - All threads perform parallel merge on input and output tiles

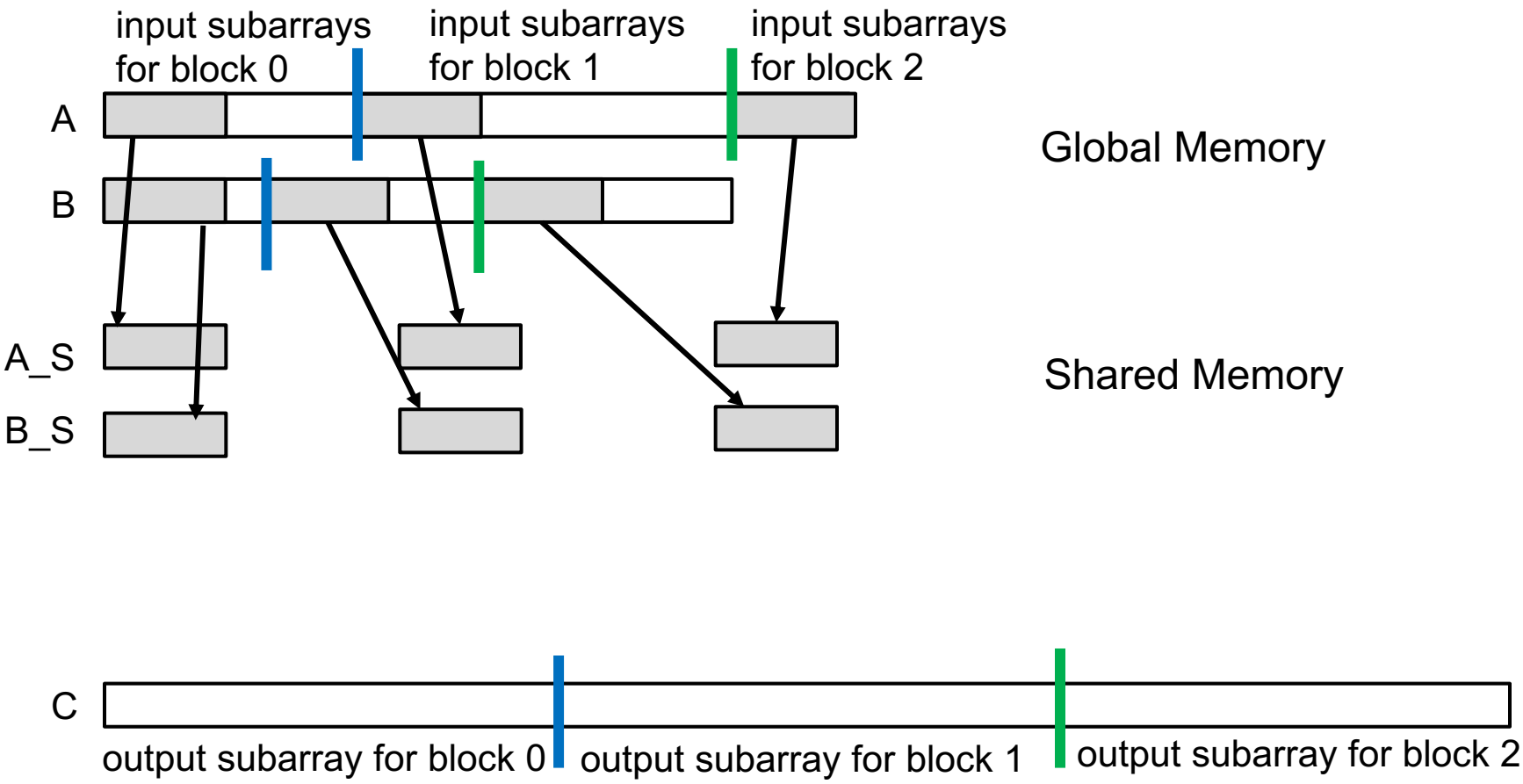
Tiled Merge Kernel: Co-Rank Values (I)



Tiled Merge Kernel: Co-Rank Values (II)

```
__global__ void merge_tiled_kernel(int* A, int m, int* B, int n, int* C, int tile_size){
  // Shared memory allocation
  extern __shared__ int shareAB[];
  int* A_S = &shareAB[0];          // A_S is the first half of shareAB
  int* B_S = &shareAB[tile_size];  // B_S is the second half of ShareAB

  // Starting point for current block
  int C_curr = blockIdx.x * ceil((m+n)/gridDim.x);
  // Starting point for next block
  int C_next = min((blockIdx.x+1) * ceil((m+n)/gridDim.x), (m+n));

  if (threadIdx.x == 0)
  {
    A_S[0] = co_rank(C_curr, A, m, B, n); // Make the block-level co-rank values visible to
    A_S[1] = co_rank(C_next, A, m, B, n); // other threads in the block
  }
  __syncthreads();

  int A_curr = A_S[0]; int B_curr = C_curr – A_curr;
  int A_next = A_S[1]; int B_next = C_next – A_next;

  __syncthreads();

...
```

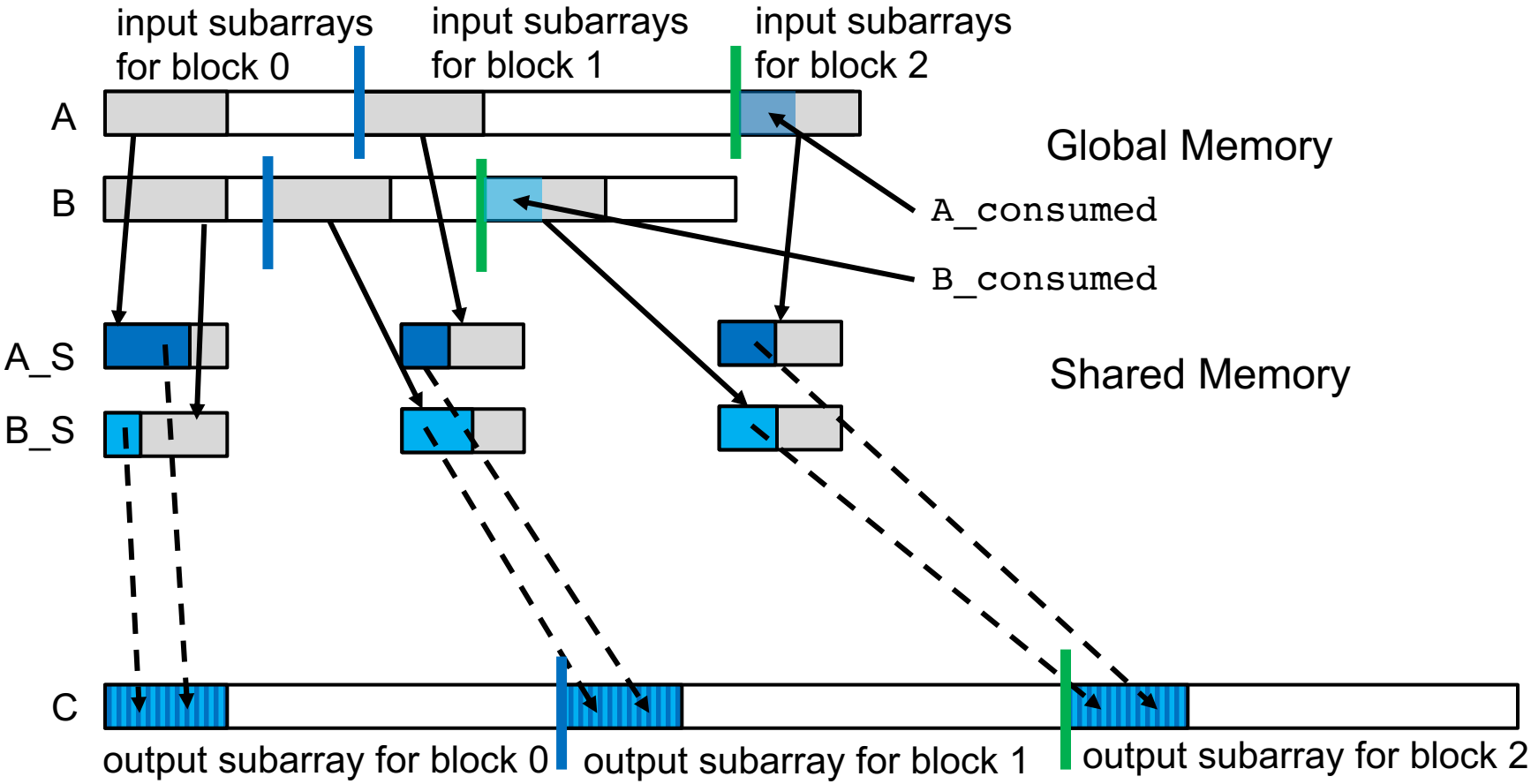
Tiled Merge Kernel: Loading Tiles (Iteration 0)



Tiled Merge Kernel: Loading Tiles

```
int counter = 0;  // Iteration counter
int C_length = C_next – C_curr;
int A_length = A_next – A_curr;
int B_length = B_next – B_curr;
int total_iteration = ceil((C_length)/tile_size);  // Total iterations
int C_completed = 0;   int A_consumed = 0;   int B_consumed = 0;

while(counter < total_iteration)
{
  // Loading tile-size A and B elements into shared memory
  for(int i=0; i<tile_size; i+=blockDim.x)
  {
    if(i + threadIdx.x < A_length – A_consumed)
    {
      A_S[i + threadIdx.x] = A[A_curr + A_consumed + i + threadIdx.x];
    }
  }

  for(int i=0; i<tile_size; i+=blockDim.x)
  {
    if(i + threadIdx.x  < B_length – B_consumed)
    {
      B_S[i + threadIdx.x] = B[B_curr + B_consumed + i + threadIdx.x];
    }
  }

  __syncthreads();

...
```

Tiled Merge Kernel: Generating Output Tiles (Iteration 0)



Tiled Merge Kernel: Partitioning Tiles among Threads

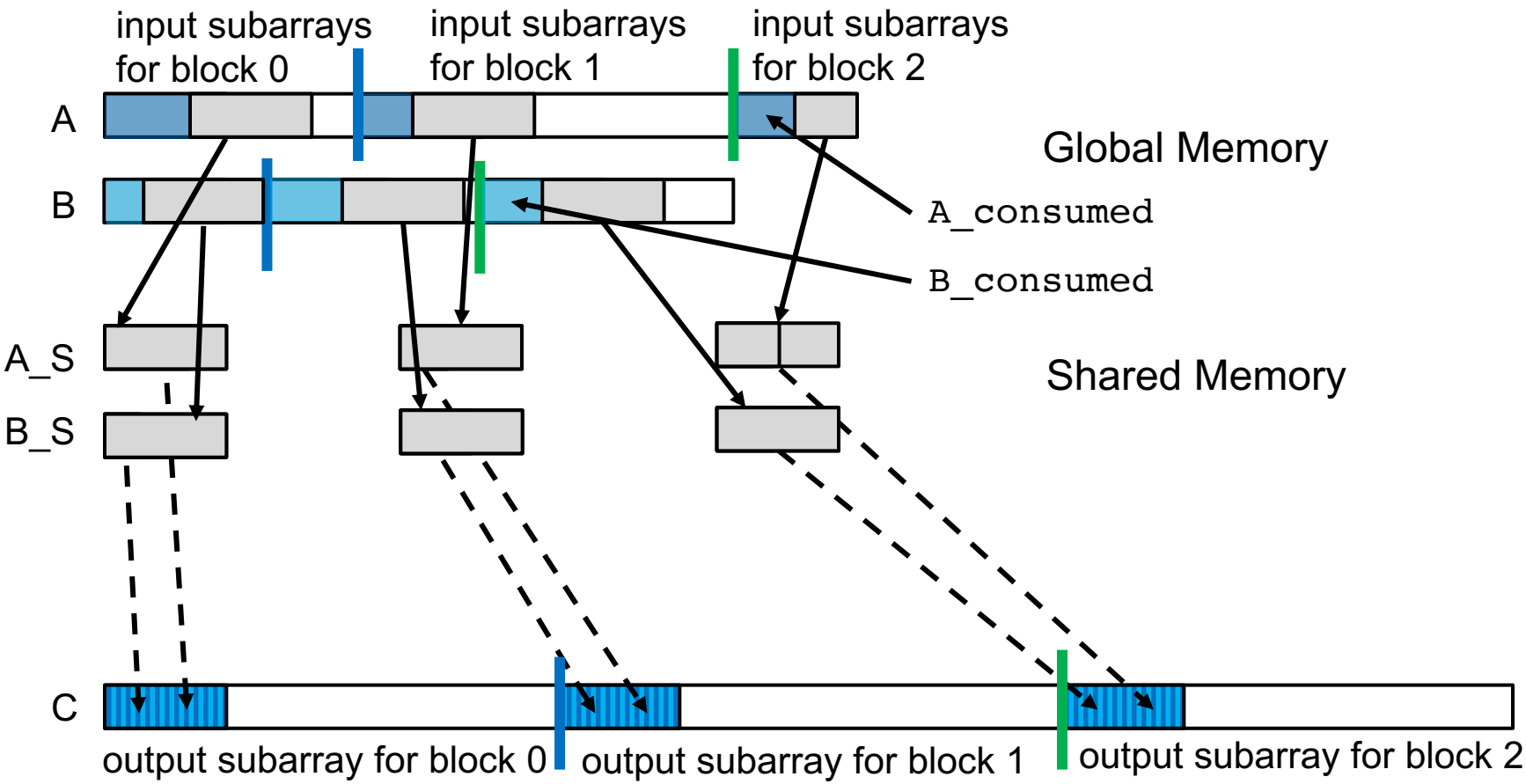
```
...

int c_curr = threadIdx.x  *  (tile_size/blockDim.x);
int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);


c_curr = (c_curr <= C_length – C_completed) ? c_curr : C_length – C_completed;
c_next = (c_next <= C_length – C_completed) ? c_next : C_length – C_completed;


// Find co-rank for c_curr and c_next
int a_curr = co_rank(c_curr, A_S, min(tile_size, A_length-A_consume),
                                  B_S, min(tile_size, B_length-B_consumed));
int b_curr = c_curr – a_curr;

int a_next = co_rank(c_next, A_S, min(tile_size, A_length-A_consumed),
                                  B_S, min(tile_size, B_length-B_consumed));
int b_next = c_next – a_next;

...
```

Each Thread Generates a Subsection of the Output Tile

```
  ...

    // All threads call sequential merge
    merge_sequential(A_S+a_curr, a_next-a_curr, B_S+b_curr, b_next-b_curr,
                           C+C_curr+C_completed+c_curr);

    // Update the A and B elements that have been consumed thus far
    counter ++;
    C_completed += tile_size;
    A_consumed += co_rank(tile_size,  A_S, tile_size, B_S, tile_size);
    B_consumed = C_completed – A_consumed;

    __syncthreads();
  }
}
```

Performance can be further improved by generating output to shared memory first

Tiled Merge Kernel: Loading Tiles (Iteration 1)

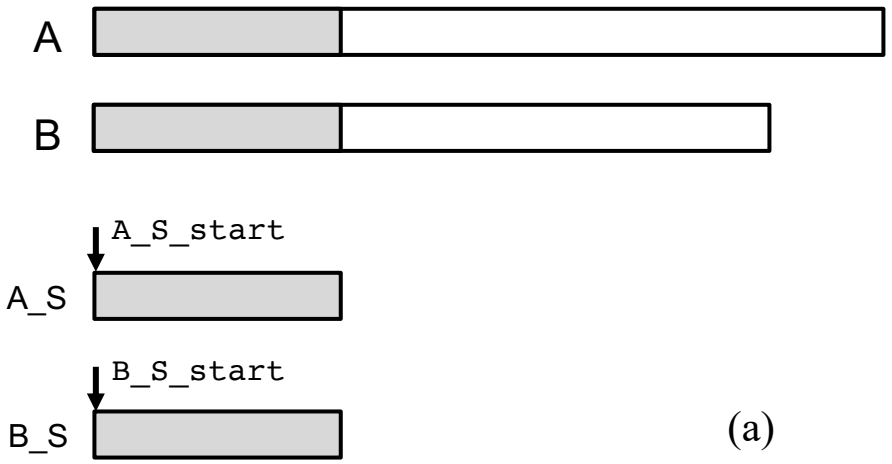


Tiled Merge Kernel: Pros & Cons

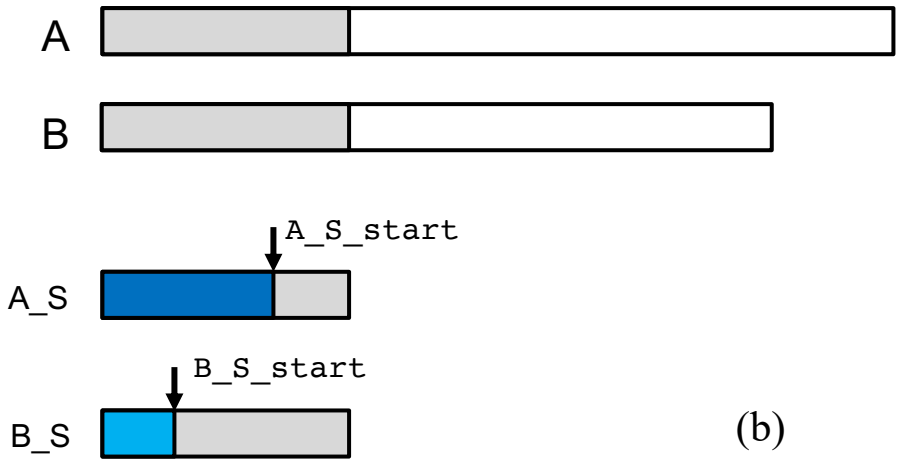
- Pros
  - Coalesced loads of the input elements
  - Reduced global memory traffic for co-rank functions
    - Thread-level co-rank functions are done in shared memory
  - Coalesced stores if the output tiles are generated in shared memory

- Cons
  - Only half of the input elements loaded into shared memory are used (in the worst case)

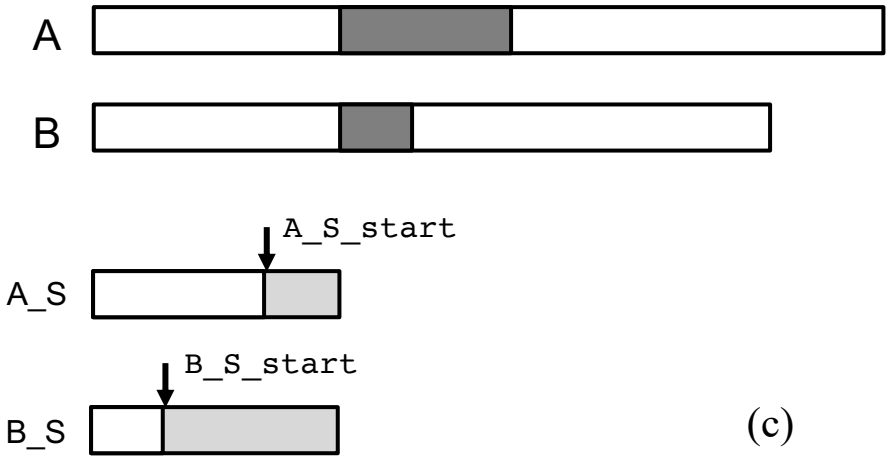
Circular Buffering



(a)



(b)



(c)

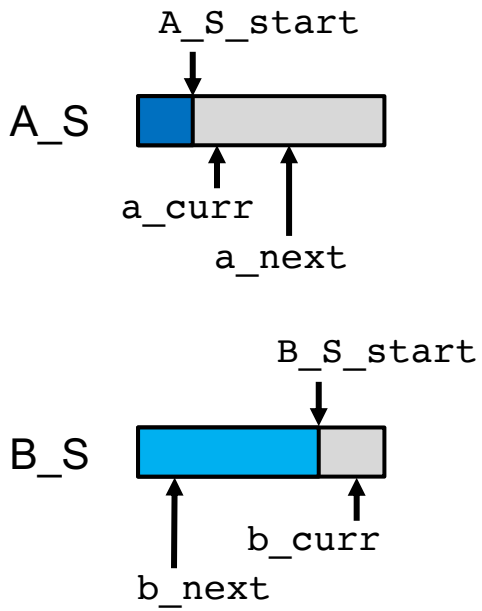
(d)

Loading Circular Buffering Tiles

```
...

int A_S_start = 0;
int B_S_start = 0;
int A_S_consumed = tile_size;  // In the first iteration, fill the tile_size
int B_S_consumed = tile_size;  // In the first iteration, fill the tile_size

while(counter < total_iteration)
{
  // Loading (refilling) A_S_consumed elements into A_S
  for(int i=0; i<A_S_consumed; i+=blockDim.x){
    if(i + threadIdx.x < A_length - A_consumed && i + threadIdx.x < A_S_consumed)
    {
      A_S[(A_S_start + (tile_size-A_S_consumed) + i + threadIdx.x)%tile_size] =
                                                    A[A_curr + A_consumed + i + threadIdx.x];
    }
  }

  // Loading B_S_consumed elements into B_S
  for(int i=0; i<B_S_consumed; i+=blockDim.x){
    if(i + threadIdx.x  < B_length - B_consumed && i + threadIdx.x < B_S_consumed)
    {
      B_S[(B_S_start + (tile_size-B_S_consumed + i + threadIdx.x)%tile_size] =
                                                    B[B_curr + B_consumed + i + threadIdx.x];
    }
  }

...
```

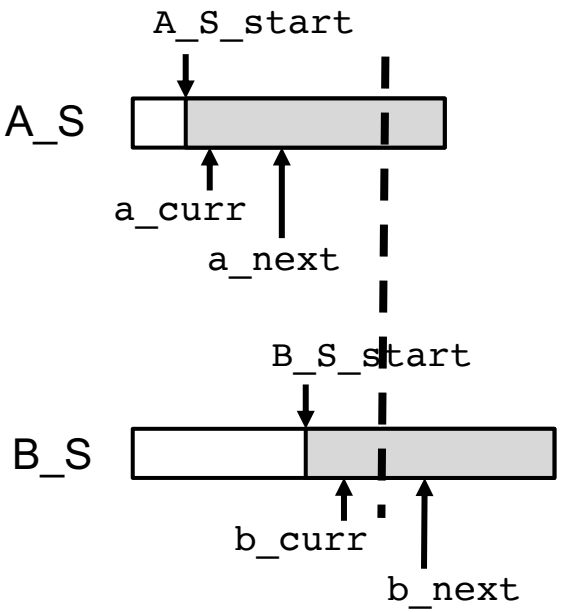
Simplified Model for Co-Rank Values

- Tiles wrap around in the circular buffer
  - This makes the handling of co-rank values more complex



(a) Reality

```
a_next – a_curr
b_next – b_curr + tile_size
```



(b) Simplified

```
a_next – a_curr
b_next – b_curr
```

Circular-Buffer Merge Kernel (I)

```
...

    int c_curr = threadIdx.x  *  (tile_size/blockDim.x);
    int c_next = (threadIdx.x+1) * (tile_size/blockDim.x);

    c_curr = (c_curr <= C_length-C_completed) ? c_curr : C_length-C_completed;
    c_next = (c_next <= C_length-C_completed) ? c_next : C_length-C_completed;

    // Find co-rank for c_curr and c_next
    int a_curr = co_rank_circular(c_curr,
                                  A_S, min(tile_size, A_length-A_completed),
                                  B_S, min(tile_size, B_length-B_completed),
                                  A_S_start, B_S_start, tile_size);
    int b_curr = c_curr - a_curr;

    int a_next = co_rank_circular(c_next,
                                  A_S, min(tile_size, A_length-A_completed),
                                  B_S, min(tile_size, B_length-B_completed),
                                  A_S_start, B_S_start, tile_size);
    int b_next = c_next - a_next;

    // Do merge in parallel
    merge_sequential_circular(A_S, a_next-a_curr, B_S, b_next-b_curr,
                              C+C_curr+C_completed+c_curr,
                              A_S_start+a_curr, B_S_start+b_curr, tile_size);

...
```

Circular-Buffer Merge Kernel (II)

```
...

    // Figure out the work has been done
    counter++;
    A_S_consumed = co_rank_circular(min(tile_size, C_length-C_completed),
                                    A_S, min(tile_size, A_length-A_consumed),
                                    B_S, min(tile_size, B_length-B_consumed),
                                    A_S_start, B_S_start, tile_size);

    B_S_consumed = min(tile_size, C_length-C_completed) - A_S_consumed;

    A_consumed += A_S_consumed;

    C_completed += min(tile_size, C_length-C_completed);

    B_consumed = C_completed - A_consumed;

    A_S_start = A_S_start + A_S_consumed;
    if(A_S_start >= tile_size) A_S_start = A_S_start - tile_size;

    B_S_start = B_S_start + B_S_consumed;
    if(B_S_start >= tile_size) B_S_start = B_S_start - tile_size;

    __syncthreads();
  }
}
```

Sequential merge and co-rank functions are the only change:
A well-designed library interface limits the impact on user code

Co-Rank Function with Circular Buffer

```
int co_rank_circular(int k, int* A, int m, int* B, int n, int A_S_start, int B_S_start, int tile_size){
  int i= k<m ? k : m;  // i = min(k, m)
  int j = k - i;
  int i_low = 0>(k-n) ? 0 : k-n;  // i_low = max(0, k-n)
  int j_low = 0>(k-m) ? 0: k-m;   // i_low = max(0, k-m)
  int delta;
  bool active = true;
  while(active){
    int i_cir = (A_S_start+i) % tile_size;
    int i_m_1_cir = (A_S_start+i-1) % tile_size;
    int j_cir = (B_S_start+j) % tile_size);
    int j_m_1_cir = (B_S_start+i-1) % tile_size;

    if(i > 0 && j < n && A[i_m_1_cir] > B[j_cir]){
      delta = ((i - i_low +1) >> 1); // ceil(i-i_low)/2)
      j_low = j;
      i = i - delta;
      j = j + delta;
    } else if(j > 0 && i < m && B[j_m_1_cir] >= A[i_cir]){
      delta = ((j - j_low +1) >> 1);
      i_low = i;
      i = i + delta;
      j = j - delta;
      } else{
      active = false;
    }
  }
  return i;
}
```

Sequential Merge with Circular Buffers

```
void merge_sequential_circular(int *A, int m, int *B, int n, int *C, int A_S_start, int B_S_start, int tile_size){
  int i = 0;  // Virtual index into A
  int j = 0;  // Virtual index into B
  int k = 0;  // virtual index into C

  while((i < m) && (j < n)){
    int i_cir = (A_S_start + i) % tile_size;
    int j_cir = (B_S_start + j) % tile_size;

    if(A[i_cir] <= B[j_cir]){
      C[k++] = A[i_cir]; i++;
    } else{
      C[k++] = B[j_cir]; j++;
    }
  }

  if(i == m) {  // Done with A[] handle remaining B[]
    for(; j < n; j++) {
      int j_cir = (B_S_start + j) % tile_size);
      C[k++] = B[j_cir];
    }
  } else{       // Done with B[], handle remaining A[]
    for(; i <m; i++) {
      int i_cir = (A_S_start + i) % tile_size;
      C[k++] = A[i_cir];
    }
  }
}
```

Merge Sort: Summary

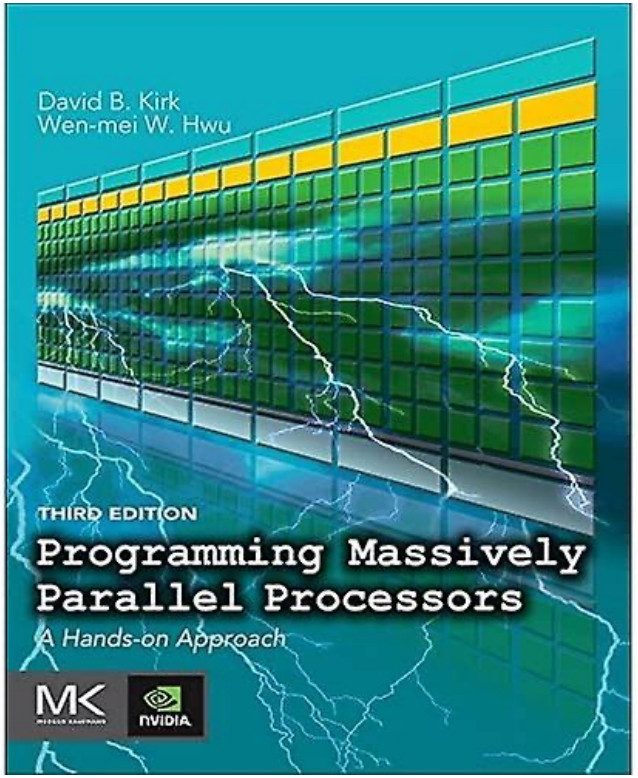
- Parallelization of merge sort requires each thread to dynamically identify its input ranges
  - Input ranges are data dependent
  - Challenges when using tiling

- Circular buffers to make full use of the data loaded into shared memory
  - Simplified buffer access model to not increase code complexity

Recommended Readings (I)

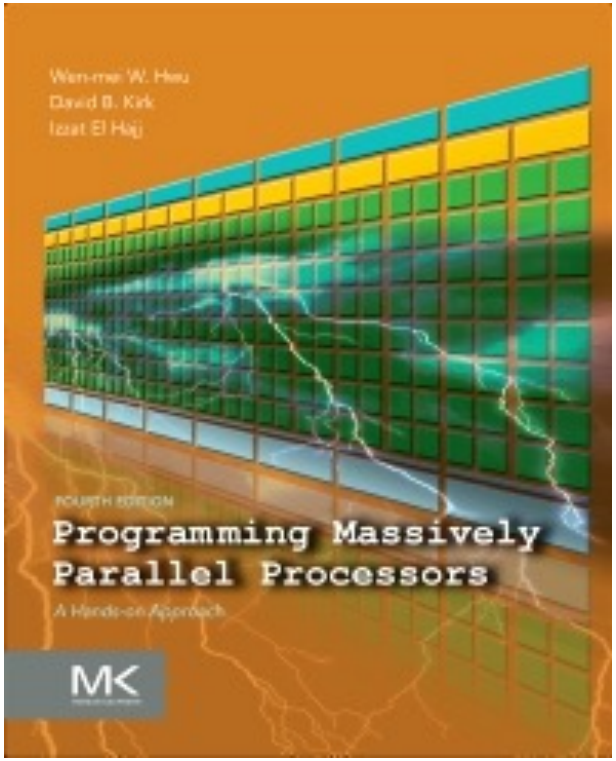
- Hwu and Kirk, "Programming Massively Parallel Processors," Third Edition, 2017
  - Chapter 11 - Parallel patterns:

merge sort: An introduction to tiling with dynamic input data identification



Recommended Readings (II)

- Hwu and Kirk and El Hajj, "Programming Massively Parallel Processors," Fourth Edition, 2022
  - Chapter 12 - Merge: An introduction to dynamic input data identification



P&S Heterogeneous Systems

Parallel Patterns: Merge Sort

Dr. Juan Gómez Luna
Prof. Onur Mutlu
ETH Zürich

Fall 2022

2 January 2023